AS/400 Advanced Series

IBM Access Class Library for Windows** Reference

Version 3

SC41-4622-00

Chapter 3. Database Classes

The database access classes are a set of classes that use the Open Database Connectivity (ODBC) interface to access data in the AS/400 database.

Many of the member functions defined in the database access classes use ODBC APIs. As a result, the member functions may have parameters that accept predefined constant values. For detailed information on how to use these database access member functions including possible values for parameters, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

The Database Access Classes can be used with AS/400 Client Access for Windows 95/NT.

The following tables list the ODBC API used by the database access member functions.

Table 1. IC4SQLDatabase

Member Function	**ODBC API**
columnPrivileges	SQLColumnPrivileges
columns	SQLColumns
commit	SQLTransact
connect	SQLConnect
connectOption	SQLGetConnectOption
disconnect	SQLDisconnect
foreignKeys	SQLForeignKeys
functionSupported	SQLGetFunctions
getInformation	SQLGetInfo
getStatusInformation	SQLError
nativeSql	SQLNativeSql
primaryKeys	SQLPrimaryKeys
procedureColumns	SQLProcedureColumns
procedures	SQLProcedures
rollback	SQLTransact
setConnectOption	SQLSetConnectOption
specialColumns	SQLSpecialColumns
statistics	SQLStatistics
tablePrivileges	SQLTablesPrivileges
tables	SQLTables
typeInformation	SQLGetTypeInfo

Database Classes

Table 2. IC4SQLEnvironment

Member Function	**ODBC API**
commitDatabases	SQLTransact
datasources	SQLDataSources
getStatusInformation	SQLError
rollbackDatabases	SQLTransact

Table 3. IC4SQLResultSet

Member Function	**ODBC API**
bindColumn	SQLBindCol
bindColumns	SQLBindCol
bindToRow	SQLBindCol
close	SQLFreeStmt
columnAttribute	SQLColAttributes
cursorName	SQLGetCursorName
describeColumn	SQLDescribeCol
extendedFetch	SQLExtendedFetch
fetch	SQLFetch
getData	SQLGetData
getStatusInformation	SQLError
moreResults	SQLMoreResults
numberResultColumns	SQLNumResultCols
parameterData	SQLParamData
putData	SQLPutData
rowAsString	SQLGetData
rowsAffected	SQLRowCount
setPosition	SQLSetPos
unbind	SQLFreeStmt

Table 4 (Page 1 of 2). IC4SQLStatement

Member Function	**ODBC API**
bindParameter	SQLSetParam • ODBC version 1.0
bindParameter (parameter type; max length)	SQLBindParameter • ODBC version 2.0
bindParameters	SQLSetParam
cancel	SQLCancel

Table 4 (Page 2 of 2). IC4SQLStatement

Member Function	**ODBC API**
execute	SQLExecute
executeDirect	SQLExecDirect
getParameterDescription	SQLDescribeParam
getStatusInformation	SQLError
`IC4SQLStatement` constructors	SQLAllocStmt
moreResults	SQLMoreResults
numberParameters	SQLNumParams
parameterData	SQLParamData
prepare	SQLPrepare
putData	SQLPutData
rowsAffected	SQLRowCount
setCursorName	SQLSetCursorName
setParameterOptions	SQLParamOptions
setStatementOption	SQLSetStmtOption SQLSetScrollOptions
statementOption	SQLGetStmtOption
unbindParameters	SQLFreeStmt
unprepare	SQLFreeStmt

IC4SQLDatabase

The `IC4SQLDatabase` class represents a connection to the database on an AS/400 system. Objects of this class manage a connection to an AS/400 system and provide catalog functions that can be used to query information about the AS/400 database.

Derivation

`IC4SQLDatabase` does not derive from another class.

Header File

`ic4sqdb.hpp`

Constructors

`IC4SQLDatabase``(const IC4SQLEnvironment& `*`environment`*`);`

The parameter is the following:

environment	The IC4SQLEnvironment object that was instantiated for the application.

Exceptions

- IC4SQLException

Public Members

columnPrivileges

```
virtual RETCODE columnPrivileges(IC4SQLResultSet& results,
                                 const unsigned char* tableQualifier,
                                 short tableQualifierLength,
                                 const unsigned char* tableOwner,
                                 short tableOwnerLength,
                                 const unsigned char* tableName,
                                 short tableNameLength,
                                 const unsigned char* columnName,
                                 short columnNameLength);
```

Generates a result set with information about the columns in a table that satisfy certain criteria.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.
tableQualifier	(in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
tableQualifierLength	(in) Length of table qualifier.
tableOwner	(in) Name of table owner. For an AS/400 system, this is the library name.
tableOwnerLength	(in) Length of owner name.
tableName	(in) Name of table for which information is to be retrieved.
tableNameLength	(in) Length of table name.
columnName	(in) String search pattern for column names.
columnNameLength	(in) Length of string search pattern.

API used: SQLColumnPrivileges

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

columns

```
virtual RETCODE columns(IC4SQLResultSet& results,
                        const unsigned char* tableQualifier,
                        short tableQualifierLength,
                        const unsigned char* tableOwner,
                        short tableOwnerLength,
                        const unsigned char* tableName,
                        short tableNameLength,
                        const unsigned char* columnName,
                        short columnNameLength);
```

Generates a result set that contains information about the columns in a table that satisfy certain criteria.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.
tableQualifier	(in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
tableQualifierLength	(in) Length of table qualifier. For an AS/400 system, this is the library name.
tableOwner	(in) Name of table owner. For an AS/400 system, this is the library name.
tableOwnerLength	(in) Length of owner name.
tableName	(in) Name of table for which information is to be retrieved.
tableNameLength	(in) Length of table name.
columnName	(in) String search pattern for column names.
columnNameLength	(in) Length of string search pattern.

API used: SQLColumns

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

commit

```
virtual RETCODE commit();
```

Commits changes initiated by all statements active under this database object.

API used: SQLTransact

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

connect

```
virtual RETCODE connect(const char* dataSource,
                        const char* userid =NULL,
                        const char* passwd =NULL);
```

Connects to a database.

The parameters are the following:

dataSource	(in) The name of the system.
userid	(in) The userid (authorization name). The value must be null terminated. This is an optional parameter. If not specified, the default userid for Client Access/400 is used.
passwd	(in) The password (authentication string). The value must be null terminated. This is an optional parameter. If not specified, the default password for Client Access/400 is used.

API used: SQLConnect

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

connectOption

```
virtual RETCODE connectOption(unsigned short option,
                              unsigned long& value);
virtual RETCODE connectOption(unsigned short option,
                              char* value);
virtual RETCODE connectOption(unsigned short option,
                              IString& value);
```

Returns the current value of a connection option. For information about connection options and the data types returned, refer to the *ODBC Programmer's Reference* for API SQLGetConnectOption.

The parameters are the following:

option	(in) The connection option to retrieve
value	(out) The value of the connection option

API used: SQLGetConnectOption

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

disconnect

```
virtual RETCODE disconnect();
```

Disconnects from the database. If commitment control is on (this is the default) and there are incomplete transactions, call `commit()` or `rollback()` first.

API used: SQLDisconnect

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

foreignKeys

```
virtual RETCODE foreignKeys(IC4SQLResultSet& results,
                       const unsigned char*     primaryKeyTableQualifier,
                       short                    primaryKeyTableQualifierLength,
                       const unsigned char*     primaryKeyTableOwner,
                       short                    primaryKeyTableOwnerLength,
                       const unsigned char*     primaryKeyTableName,
                       short                    primaryKeyTableNameLength,
                       const unsigned char*     foreignKeyTableQualifier,
                       short                    foreignKeyTableQualifierLength,
                       const unsigned char*     foreignKeyTableOwner,
                       short                    foreignKeyTableOwnerLength,
                       const unsigned char*     foreignKeyTableName,
                       short                    foreignKeyTableNameLength);
```

Generates a result set of information about foreign keys in the specified table or in tables that refer to the primary keys in the specified table.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.

primaryKeyTableQualifier	(in) Primary key table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
primaryKeyTableQualifierLength	(in) Length of primary key table qualifier
primaryKeyTableOwner	(in) Name of primary key table owner
primaryKeyTableOwnerLength	(in) Length of primary key table owner name
primaryKeyTableName	(in) Primary key table name
primaryKeyTableNameLength	(in) Length of primary key table name
foreignKeyTableQualifier	(in) Foreign key table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
foreignKeyTableQualifierLength	(in) Length of foreign key table qualifier
foreignKeyTableOwner	(in) Name of foreign key table owner
foreignKeyTableOwnerLength	(in) Length of foreign key table owner name
foreignKeyTableName	(in) Foreign key table name
foreignKeyTableNameLength	(in) Length of foreign key table name

API used: SQLForeignKeys

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

functionSupported

```
virtual RETCODE functionSupported(unsigned short function,
                                  unsigned short* supported);
```

This member function returns a value indicating whether the ODBC driver being used supports the specified ODBC function(s).

The parameters are the following:

function	(in) The function being queried. `SQL_API_ALL_FUNCTIONS` can be used to request an array with information about all functions. When `SQL_API_ALL_FUNCTIONS` is specified, the second parameter must be a pointer to sufficient storage for an array of 100 unsigned short elements. When a value other

	than `SQL_API_ALL_FUNCTIONS` is specified, the second parameter must be a pointer to an unsigned short.
supported	(out) User-allocated storage to receive the information.

API used: SQLGetFunctions

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

getInformation

```
virtual RETCODE getInformation(short infoType,
                               void* infoValue,
                               short infoValueLength,
                               short& bytesAvailable);
```

Gets information, including data conversions, about the DBMS to which the application is currently connected.

The parameters are the following:

infoType	(in) Type of information desired. For example, specifying `SQL_FETCH_DIRECTION` returns a 32-bit mask enumerating the supported fetch direction options.
infoValue	(out) User-allocated storage to receive the information.
infoValueLength	(in) Length of *infoValue*.
bytesAvailable	(out) Actual length of information value available to return.

API used: SQLGetInfo

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

getStatusInformation

```
virtual RETCODE getStatusInformation(IString& sqlState,
                                     long& nativeError,
                                     IString& messageText);
virtual RETCODE getStatusInformation(char* sqlState,
                                     long& nativeError,
                                     char* messageText,
                                     short messageTextLength,
                                     short& bytesAvailable);
```

Returns status information associated with the last operation performed if the last operation returned `SQL_SUCCESS_WITH_INFO`. If another operation is performed with this object, the information for the previous operation can no longer be accessed. Once the information is retrieved, attempting to retrieve it again returns `SQL_NO_DATA_FOUND`. If the last operation returned `SQL_SUCCESS` or threw an exception, `SQL_NO_DATA_FOUND` is returned.

The parameters are the following:

sqlState	(out) User-allocated storage to receive the value of SQLSTATE. Must be at least 5 bytes long.
nativeError	(out) User-allocated storage to receive the native error value.
messageText	(out) User-allocated storage to receive the error text.
messageTextLength	(in) Length of messageText.
bytesAvailable	(out) Length of text available to return.

API used: SQLError

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

nativeSql

```
virtual RETCODE nativeSql(const unsigned char* inputString,
                          long inputStringLength,
                          IString& resultString);
virtual RETCODE nativeSql(const unsigned char* inputString,
                          long inputStringLength,
                          unsigned char* resultString,
                          long resultStringLength,
                          long& resultStringBytesAvailable);
```

Returns the SQL string as translated by the ODBC driver.

The parameters are the following:

inputString	(in) SQL text string to translate.
inputStringLength	(in) Length of text string to translate.
resultString	(out) User-allocated storage to receive the translated string.
resultStringLength	(in) Length of *resultString*.

resultStringBytesAvailable	(out) Length of translated string available to return.

API used: SQLNativeSql

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

primaryKeys

```
virtual RETCODE primaryKeys(IC4SQLResultSet& results,
                            const unsigned char* keyTableQualifier,
                            short keyTableQualifierLength,
                            const unsigned char* keyTableOwner,
                            short keyTableOwnerLength,
                            const unsigned char* keyTableName,
                            short keyTableNameLength);
```

Retrieves information about columns in the specified table that compose the primary key. This member function causes a result set to be generated.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.
keyTableQualifier	(in) Key table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
keyTableQualifierLength	(in) Length of key table qualifier
keyTableOwner	(in) Owner of key table
keyTableOwnerLength	(in) Length of key table owner
keyTableName	(in) Name of key table
keyTableNameLength	(in) Length of key table name

API used: SQLPrimaryKeys

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

procedureColumns

```
virtual RETCODE procedureColumns(IC4SQLResultSet& results,
                                 const unsigned char* procedureQualifier,
                                 short procedureQualifierLength,
                                 const unsigned char* procedureOwner,
                                 short procedureOwnerLength,
                                 const unsigned char* procedureName,
                                 short procedureNameLength,
                                 const unsigned char* columnName,
                                 short columnNameLength);
```

Generates a result set of information about parameters for the specified procedure and information about the columns that make up the result set for the specified procedure.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results
procedureQualifier	(in) Procedure qualifier
procedureQualifierLength	(in) Length of procedure qualifier
procedureOwner	(in) Owner of procedure
procedureOwnerLength	(in) Length of procedure owner
procedureName	(in) Name of procedure
procedureNameLength	(in) Length of procedure name
columnName	(in) String search pattern for column name
columnNameLength	(in) Length of string search pattern

API used: SQLProcedureColumns

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

procedures

```
virtual RETCODE procedures(IC4SQLResultSet& results,
                           const unsigned char* procedureQualifier,
                           short procedureQualifierLength,
                           const unsigned char* procedureOwner,
                           short procedureOwnerLength,
                           const unsigned char* procedureName,
                           short procedureNameLength);
```

Generates a result set of procedure names stored in a specific data source.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results
procedureQualifier	(in) Procedure qualifier
procedureQualifierLength	(in) Length of procedure qualifier
procedureOwner	(in) Owner of procedure
procedureOwnerLength	(in) Length of procedure owner
procedureName	(in) Name of procedure
procedureNameLength	(in) Length of procedure name

API used: SQLProcedures

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

rollback

```
virtual RETCODE rollback();
```

Rolls back changes initiated by all statements active under this database object.

API used: SQLTransact

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

setConnectOption

```
virtual RETCODE setConnectOption(const unsigned short option,
                                 const signed long value);
virtual RETCODE setConnectOption(const unsigned short option,
                                 const char* value);
```

Sets a connection option. Two versions are provided: one for specifying a long option and one for providing a char * option value. If the option specified does not correspond to the type of the value, an exception is thrown.

The parameters are the following:

option	(in) The connection option to set.

value	(in) The value of the connection option. Allowable values are an integer or character string.

API used: SQLSetConnectOption

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

specialColumns

```
virtual RETCODE specialColumns(IC4SQLResultSet& results,
                               unsigned short columnType,
                               const unsigned char* tableQualifier,
                               short tableQualifierLength,
                               const unsigned char* tableOwner,
                               short tableOwnerLength,
                               const unsigned char* tableName,
                               short tableNameLength,
                               unsigned short scope,
                               unsigned short nullable);
```

Generates a result set with information about columns in the specified table that either uniquely identify a row in the table or are automatically updated when any value in the row is changed.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.
columnType	(in) Type of information to return. Allowable values are: • `SQL_BEST_ROWID` • `SQL_ROWVER`
tableQualifier	(in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
tableQualifierLength	(in) Length of table qualifier.
tableOwner	(in) Name of table owner. For an AS/400 system, this is the library name.
tableOwnerLength	(in) Length of owner name.
tableName	(in) Name of table for which information is to be retrieved.
tableNameLength	(in) Length of table name.
scope	(in) Minimum required scope. Allowable values are: • `SQL_SCOPE_CURROW` • `SQL_SCOPE_TRANSACTION`

nullable	(in) Whether to return nullable columns. Allowable values are: • `SQL_NULLABLE` • `SQL_NO_NULLS`

API used: SQLSpecialColumns

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

statistics

```
virtual RETCODE statistics(IC4SQLResultSet& results,
                           const unsigned char* tableQualifier,
                           short tableQualifierLength,
                           const unsigned char* tableOwner,
                           short tableOwnerLength,
                           const unsigned char* tableName,
                           short tableNameLength,
                           unsigned short unique,
                           unsigned short accuracy);
```

Generates a result set with statistics about a table and the indexes associated with the table.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.
tableQualifier	(in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
tableQualifierLength	(in) Length of table qualifier.
tableOwner	(in) Name of table owner. For an AS/400 system, this is the library name.
tableOwnerLength	(in) Length of owner name.
tableName	(in) Name of table for which information is to be retrieved.
tableNameLength	(in) Length of table name.
unique	(in) Type of index. Allowable values are: • `SQL_INDEX_UNIQUE` • `SQL_INDEX_ALL`
accuracy	(in) Importance of `CARDINALITY` and `PAGES`. Allowable values are: • `SQL_ENSURE` • `SQL_QUICK`

API used: SQLStatistics

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

tablePrivileges

```
virtual RETCODE tablePrivileges(IC4SQLResultSet& results,
                                const unsigned char* tableQualifier,
                                short tableQualifierLength,
                                const unsigned char* tableOwner,
                                short tableOwnerLength,
                                const unsigned char* tableName,
                                short tableNameLength);
```

Generates a result set with a list of tables and the privileges associated with each table.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.
tableQualifier	(in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
tableQualifierLength	(in) Length of table qualifier.
tableOwner	(in) Name of table owner. For an AS/400 system, this is the library name.
tableOwnerLength	(in) Length of owner name.
tableName	(in) Name of table for which information is to be retrieved.
tableNameLength	(in) Length of table name.

API used: SQLTablePrivileges

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

tables

```
virtual RETCODE tables(IC4SQLResultSet& results,
                       const unsigned char* tableQualifier,
                       short tableQualifierLength,
                       const unsigned char* tableOwner,
                       short tableOwnerLength,
                       const unsigned char* tableName,
              short tableNameLength,
              const unsigned char* tableType,
              short tableTypeLength);
```

Generates a result set with a list of the tables in the database that match the value specified for *tableType*.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.
tableQualifier	(in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string.
tableQualifierLength	(in) Length of table qualifier.
tableOwner	(in) Name of table owner. For an AS/400 system, this is the library name.
tableOwnerLength	(in) Length of owner name.
tableName	(in) Name of table for which information is to be retrieved.
tableNameLength	(in) Length of table name.
tableType	(in) List of table types to match.
tableTypeLength	(in) Length of tableType.

API used: SQLTables

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

typeInformation

```
virtual RETCODE typeInformation(IC4SQLResultSet& results,
                                short type);
```

Generates a result set with information about data types supported by the database.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results.
type	(in) The SQL data type for which information should be returned

API used: SQLGetTypeInfo

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

IC4SQLEnvironment

The `IC4SQLEnvironment` class represents the environment that provides global functions such as committing or rolling back changes for all database objects created under the environment. An IC4SQLEnvironment object is required as input to the constructor for an IC4SQLDataBase object.

Derivation

`IC4SQLEnvironment` does not derive from another class.

Header File

`ic4sqenv.hpp`

Constructors

`IC4SQLEnvironment``();`

Only one environment can be constructed by an application.

Exceptions

- IC4SQLException

Public Members

commitDatabases

`virtual RETCODE` **`commitDatabases`**`();`

Commits all transactions active across all connected `IC4SQLDatabase` objects.

API used: SQLTransact

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

dataSources

```
virtual RETCODE dataSources(IString& dataSourceName,
                            IString& description,
                            short direction =SQL_FETCH_NEXT):
virtual RETCODE dataSources(unsigned char* dataSourceName,
                            short dataSourceNameLength,
                            short& dataSourceNameBytesAvailable,
                            unsigned char* description,
                            short descriptionLength,
                            short& descriptionBytesAvailable,
                            short direction =SQL_FETCH_NEXT);
```

Returns the next data source. Using this member function, a program can request the available data sources one at a time.

The parameters are the following:

dataSourceName	(out) User-allocated storage to receive the data source name
dataSourceNameLength	(in) Length of *dataSourceName*
dataSourceNameBytesAvailable	(out) Size of name available to return
description	(out) User-allocated storage to receive the description of data source
descriptionLength	(in) Size of *description*
descriptionBytesAvailable	(out) Size of *description* available to return
direction	(in) Direction of fetch • `SQL_FETCH_NEXT` • `SQL_FETCH_FIRST`

API used: SQLDataSources

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

environmentOption

This function is not supported.

getStatusInformation

```
virtual RETCODE getStatusInformation(IString sqlState,
                                     long& nativeError,
                                     IString& messageText);
virtual RETCODE getStatusInformation(char* sqlState,
                                     long& nativeError,
                                     char* messageText,
                                     short messageTextLength,
                                     short& bytesAvailable);
```

Returns status information associated with the last operation performed if the last operation returned `SQL_SUCCESS_WITH_INFO`. If another operation is performed with this object, the information for the previous operation can no longer be accessed. Once the information is retrieved, attempting to retrieve it again returns `SQL_NO_DATA_FOUND`. If the last operation returned `SQL_SUCCESS` or threw an exception, `SQL_NO_DATA_FOUND` is returned.

The parameters are the following:

sqlState	(out) User-allocated storage to receive the value of `SQLSTATE`. Must be at least 5 bytes long.
nativeError	(out) User-allocated storage to receive the native error value.
messageText	(out) User-allocated storage to receive the error text.
messageTextLength	(in) Length of *messageText*.
bytesAvailable	(out) Length of text available to return.

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NO_DATA_FOUND`

API used: SQLError

Exceptions

- IC4SQLException

rollbackDatabases

```
virtual RETCODE rollbackDatabases();
```

Rolls back all open transactions across all connected `IC4SQLDatabase` objects.

API used: SQLTransact

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

setEnvironmentOption

This function is not supported.

IC4SQLResultSet

The `IC4SQLResultSet` class represents a result set generated by an SQL SELECT statement or by a catalog function of the `IC4SQLDatabase` class. Member functions are available for the following:

- Bind and unbind of result columns to user variables or row objects
- Fetch rows from the result set, either one-by-one or in blocks which can be scrolled
- Setting user variables from column data from a fetched row
- Retrieve the cursor name, column attributes or number of columns
- Close the result set and deallocate resources

Derivation

`IC4SQLResultSet` does not derive from another class.

Header File

`ic4sqrs.hpp`

Constructors

```
IC4SQLResultSet();
```

Public Members

bindColumn

This member function associates (binds) a user variable to a column (field) in a result set. This enables data to be transferred from the result set to the application when a fetch is done.

```
virtual RETCODE bindColumn(unsigned short column,
                           short cType,
                           void* variable,
                           unsigned long variableLength,
                           unsigned long* bytesAvailable = NULL);
```

The parameters are the following:

column	(in) Column to bind (1-based).
cType	(in) The C language data type for the column. Data conversion can be done by specifying a type for this parameter that is different than the type of the SQL table column.

	Conversion from `SQL_C_NUMERIC` to `SQL_CHAR` is not supported. Conversion from `SQL_C_DECIMAL` to `SQL_CHAR` is not supported.
variable	(out-deferred) Variable to bind to column. When a fetch is done, the data for the column is placed into this variable. The variable storage must still be allocated at the time the fetch is done.
variableLength	(in) Length of data that can be returned in the variable.
bytesAvailable	(out-deferred) Length of data returned after a fetch is done. This value could be `SQL_NULL_DATA`. The variable storage must still be allocated at the time the fetch is done. A run time error occurs if the default is used and the column has a `NULL` value.

Exceptions

- IC4SQLException

To have the column type determined by the type of the user variable provided, use one of these versions:

```
virtual RETCODE bindColumn(unsigned short column,
                           short& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           long& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           double& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           float& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           DATE_STRUCT& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           TIME_STRUCT& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           TIMESTAMP_STRUCT& variable,
                           unsigned long* bytesAvailable = NULL);
```

The parameters are the following:

column	(in) Column to bind.
variable	(out-deferred) Variable to bind to column. When a fetch is done, the data for the column is placed into this variable. The variable storage must still be allocated at the time the fetch is done.
bytesAvailable	(out-deferred) Size of data returned after a fetch is done. This value could be `SQL_NULL_DATA`. The variable storage must still be allocated at the time the fetch is done. A run time error occurs if the default is used and the column has a `NULL` value.

Exceptions

- IC4SQLException

To bind a column to a character buffer, use this version:

```
virtual RETCODE bindColumn(unsigned short column,
                           char* variable,
                           long variableLength,
                           unsigned long* bytesAvailable = NULL);
```

The parameters are the following:

column	(in) Column to bind.
variable	(out-deferred) Variable to bind to column. When a fetch is done, the data for the column is placed into this variable. The variable storage must still be allocated at the time the fetch is done.
variableLength	(in) Maximum length of data that can be returned in the variable.
bytesAvailable	(out-deferred) Size of data returned after a fetch is done. This value could be `SQL_NULL_DATA`. The variable storage must still be allocated at the time the fetch is done. A run time error occurs if the default is used and the column has a `NULL` value.

To bind a column to a variable object, use this version:

```
virtual RETCODE bindColumn(const IC4SQLVariable& variable);
```

The input variable object must contain a non-zero position number, indicating which column is to be bound to the variable. If the column is already bound, the previous binding is released.

The parameters are the following:

variable	(in) A variable object that describes a user variable to bind.

API used: SQLBindCol

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

bindColumns

```
virtual RETCODE bindColumns(const IC4SQLVariableList& variableList);
```

Binds multiple variables to columns using an IC4SQLVariableList. Each variable object in the list must contain a non-zero position number, indicating which column is to be bound to the variable. If the column is already bound, the previous binding is released.

The parameters are the following:

variablelist	(in) The list of user variables.

API used: SQLBindCol

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

bindToRow

```
virtual RETCODE bindToRow(IC4SQLRow& row);
```

Binds each column of the result set to a row. The row object manages the storage for each column as needed. If any columns were bound previously, the previous binding is released.

The parameters are the following:

row	(in) The row object to which columns will be bound.

API used: SQLBindCol

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

close

```
virtual RETCODE close();
```

Closes the result set. Any columns in the result set that were bound are unbound, so that the next time this result object is used, there are no bound columns initially.

API used: SQLFreeStmt

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

columnAttribute

```
virtual RETCODE columnAttribute(unsigned short column,
                                unsigned short attributeType,
                                char* characterAttribute,
                                unsigned short characterAttributeLength,
                                unsigned short& bytesAvailable,
                                long& integerAttribute);
```

Retrieves one attribute of one column. This member function can also be used to determine the number of columns by specifying 0 as the column number.

The parameters are the following:

column	(in) Column number (1 is leftmost).
attributeType	(in) Type of information. For example, `SQL_COLUMN_NAME`.
characterAttribute	(out) User-allocated storage in which a string column attribute is returned.
characterAttributeLength	(in) Length of *characterAttribute*.
bytesAvailable	(out) Actual length of the data. If *bytesAvailable* is larger than *characterAttributeLength*, then truncation occurred.
integerAttribute	(out) User-allocated storage in which a numeric column attribute is returned.

API used: SQLColAttributes

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

cursorName

```
virtual RETCODE cursorName(char* cursorName,
                           short cursorNameLength,
                           short& bytesAvailable);
virtual RETCODE cursorName(IString& cursorName);
```

Returns the name of the cursor associated with this result set.

The parameters are the following:

cursorName	(out) User-allocated storage to receive the cursor name.
cursorNameLength	(in) Length of *cursorName*.
bytesAvailable	(out) Length of the cursor name available to return. If *bytesAvailable* is larger than *cursorNameLength*, then truncation occurred.

API used: SQLGetCursorName

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

describeColumn

```
virtual RETCODE describeColumn(unsigned short column,
                               char* columnName,
                               unsigned short columnNameLength,
                               unsigned short& bytesAvailable,
                               short& sqlType,
                               unsigned long& precision,
                               short& scale,
                               short& nullable);
virtual RETCODE describeColumn(unsigned short column,
                               IString& columnName,
                               short& sqlType,
                               unsigned long& precision,
                               short& scale,
                               short& nullable);
```

Returns the result descriptor information (column name, type, precision, for example) for the indicated column.

The parameters are the following:

column	(in) Column number
columnName	(out) String to receive the column name.
columnNameLength	(in) Length of *columnName*.
bytesAvailable	(out) Length of column name available to return. If *bytesAvailable* is larger than *columnNameLength*, then truncation occurred.
sqlType	(out) SQL datatype to which the user value should be converted.
precision	(out) The precision of the column.

scale	(out) For numeric columns, the maximum number of decimal positions.
nullable	(out) Whether `NULLs` are allowed for this field. Possible values are `SQL_NO_NULLS` or `SQL_NULLABLE`.

API used: SQLDescribeCol

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

extendedFetch

```
virtual RETCODE extendedFetch(unsigned short fetchType,
                              long rowNumber,
                              unsigned long* numberRowsFetched,
                              unsigned short* rowStatus);
```

Use this member function to do blocked row processing. If this result set was generated by executing an SQL statement, the scrolling attributes and block size of this result set are taken from the statement object used to produce this result set. See `setStatementOption()` of IC4SQLStatement. If this result set was generated by a catalog function, the scrolling attributes and block size of this result set are taken from the database object used to produce this result set. See `setConnectOption()` of IC4SQLDatabase.

If you use this member function, do not use `fetch()` on the same result set.

The parameters are the following:

fetchType	(in) Type of fetch to do
rowNumber	(in) Number of the row to fetch
numberRowsFetched	(out) Actual number of rows fetched
rowStatus	(out) Array of status values

APIs used: SQLExtendedFetch

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

IC4SQLResultSet

fetch

```
virtual RETCODE fetch();
```

Fetches the next row. If you use this member function, do not use `extendedFetch()` on the same result set.

API used: SQLFetch

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

getData

Gets data for a fetched column. Stores (and converts if necessary) the data for the column into the variable.

If the columns have been bound to a user variable by using one of the `bindColumn()` methods, `getData()` will fail and an exception will be thrown.

```
virtual RETCODE getData(unsigned short column,
                        short cType,
                        void* value,
                        unsigned long valueLength,
                        unsigned long* bytesAvailable);
```

The parameters are the following:

column	(in) Column number
cType	(in) The C language type
value	(out) Buffer to receive the data.
valueLength	(in) Length of *value*
bytesAvailable	(out) Actual length of the data available to return. If *bytesAvailable* is larger than *valueLength*, then truncation occurred.

When some parameters can be determined by the type of the variable, use one of the following versions:

```
virtual RETCODE getData(unsigned short column,
                        short& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        long& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        double& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        float& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        char* variable,
                        unsigned long max,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        IString& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        DATE_STRUCT& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        TIME_STRUCT& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        TIMESTAMP_STRUCT& variable,
                        unsigned long* bytesAvailable =NULL);
```

The parameters are the following:

column	(in) Column to retrieve.
variable	(out) Variable to receive the data.
max	(in) Length of *variable*.
bytesAvailable	(out) Length of data returned in *variable* (could be SQL_NULL_DATA).

To perform `getData()` against all variables in a variable list, use this version:

```
virtual RETCODE getData(const IC4SQLVariableList& varlist);
```

Each variable object in the variable list must contain a non-zero column number. The data from that column is returned in the variable.

API used: SQLGetData

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

getStatusInformation

```
virtual RETCODE getStatusInformation(IString& sqlState,
                                     long& nativeError,
                                     IString& messageText);
virtual RETCODE getStatusInformation(char* sqlState,
                                     long& nativeError,
                                     char* messageText,
                                     short messageTextLength,
                                     short& bytesAvailable);
```

Returns status information associated with the last operation performed if the last operation returned `SQL_SUCCESS_WITH_INFO`. If another operation is performed with this object, the information for the previous operation can no longer be accessed. Once the information is retrieved, attempting to retrieve it again returns `SQL_NO_DATA_FOUND`. If the last operation returned `SQL_SUCCESS` or threw an exception, `SQL_NO_DATA_FOUND` is returned.

The parameters are the following:

sqlState	(out) User-allocated storage to receive the value of SQLSTATE. Must be at least 5 bytes long.
nativeError	(out) User-allocated storage to receive the native error value.
messageText	(out) User-allocated storage to receive the error text.
messageTextLength	(in) Length of messageText.
bytesAvailable	(out) Length of text available to return.

API used: SQLError

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NO_DATA_FOUND`

moreResults

```
virtual RETCODE moreResults();
```

Initializes processing for the next result set if more result sets are available.

API used: SQLMoreResults

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

numberResultColumns

`virtual RETCODE **numberResultColumns**(unsigned short& *columns*);`

Retrieves the number of columns in the result set.

API used: SQLNumResultCols

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

operator<<

`ostream& **operator<<**(ostream& strm, IC4SQLResultSet& *curs*);`

Writes the specified result set as a string to the specified stream.

The parameters are the following:

stream	(in) The stream to write to
curs	(in) The result set object to write

Exceptions

- None

parameterData

`virtual RETCODE **parameterData**(void* * *value*);`

If `setPosition()` has been invoked to do an add or update operation, and data-at-execution columns were specified, use this member function with `putData()` to supply column data needed to complete the operation. It returns the value that is used to find the data that will be provided as a parameter on an `execute()`.

This member function requires ODBC version 2.0.

The parameters are the following:

value	(out) Pointer to the user buffer specified on `setParameter()`.

API used: SQLParamData

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

- `SQL_STILL_EXECUTING`
- `SQL_NEED_DATA`

Exceptions

- IC4SQLException

putData

```
virtual RETCODE putData(const void* value,
                        long int length);
```

Use this member function with `parameterData()` to supply column data at statement execution time when `setPosition()` has been invoked. This member function requires ODBC version 2.0. This member function requires DB2 SQL call level interface (CLI).

The parameters are the following:

value	(in) Parameter data being provided.
length	(in) Length of parameter data.

API used: SQLPutData

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

rowAsString

```
virtual RETCODE rowAsString(IString& string);
virtual IString rowAsString();
```

Formats the current row as a string by performing a `getData()` on all the columns and building a string. Any columns that were bound are omitted from the resulting string. If a data mapping error has occurred for a column, "++++" is substituted for the column's value in the string.

API used: SQLGetData

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

rowsAffected

```
virtual RETCODE rowsAffected(long& rows);
```

Returns the number of rows in a result set that were affected by an option of update, insert, or delete done by `setPosition()`.

The parameter is the following:

rows	(out) The number of rows in a result set that were affected.

API used: SQLRowCount

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

setPosition

```
virtual RETCODE setPosition(unsigned short rowIndex,
                            unsigned short operation,
                            unsigned short lock);
```

Sets cursor position within a fetched block of data and allows a refresh to be performed. Update, delete, or add operations can be specified. This member function is used only with `extendedFetch()`.

The parameters are the following:

rowIndex	(in) Position of the row.
operation	(in) Operation to perform.
lock	(in) How to lock the row after performing the operation.

API used: SQLSetPos

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

unbind

`virtual RETCODE` **`unbind`**`();`

Unbinds all bound columns.

API used: SQLFreeStmt

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

IC4SQLRow

The `IC4SQLRow` class allows a program to fetch and save an entire row of data and perform formatting functions on the contents of the row.

Derivation

`IC4SQLRow` does not derive from another class.

Header File

`ic4sqrow.hpp`

Constructors

```
IC4SQLRow();
IC4SQLRow(IC4SQLRow const & row);
```

Public Members

asString

`IString` **`asString`**`();`

Returns a formatted string version of the row data. If a data mapping error has occurred for a column, "++++" is substituted for the column's value in the string.

columnData

```
const void* columnData(short column,
                       short& cType,
                       long& bytesAvailable);
```

A void pointer to the actual data as stored. If the row was never fetched into, then `NULL` is returned. If a data mapping error has occurred for the column, an `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 1405 in the first line of text.

The parameters are the following:

column	(in) The number of the column for which data should be returned.
cType	(out) The C language data type of the data. For example, `SQL_C_CHAR`.
bytesAvailable	(out) The actual size of the data.

Return Value: A void pointer to the actual data as stored. If the row was never fetched into, then `NULL` is returned.

Exceptions

- IC4SQLException

columnDataAsString

```
IString columnDataAsString(short column);
```

The parameter is the following:

column	(in) The column number.

Returns the data of one column stored in the row as a string. If a data mapping error has occurred for the column, an `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 1405 in the first line of text.

Exceptions

- IC4SQLException

numberOfColumns

```
long numberOfColumns() const;
```

Returns the number of elements in the row.

operator=

```
IC4SQLRow& operator=(const IC4SQLRow& row);
```

Assignment operator.

The parameter is the following:

row	The IC4SQLRow object being assigned from.

operator<<

```
ostream& operator<<(ostream& stream, const IC4SQLRow& row);
```

Writes the specified row as a string to the specified stream.

The parameters are the following:

stream	(in) The stream to write to
row	(in) The row to write

Exceptions

- None

IC4SQLStatement

The `IC4SQLStatement` class represents an SQL statement. This class handles the preparation and execution of SQL statements. This includes the following:

- Set parameters (bind a parameter marker to a user variable)
- Prepare an SQL text string
- Execute a prepared SQL text
- Execute an SQL text directly, without preparation
- Set and retrieve statement attributes that affect the behavior of statements and the result sets that are generated.

Derivation

`IC4SQLStatement` does not derive from another class.

Header File

`ic4sqst.hpp`

Constructors

```
IC4SQLStatement(IC4SQLDatabase& database);
IC4SQLStatement(IC4SQLDatabase& database,
                const char* statement);
```

The parameters are the following:

database	(in) The database object with which this statement is associated. The database object must have had a `connect()` done.
statement	(in) The text of the SQL statement to be prepared.

You can construct objects of this class in the following ways:

- Create a statement without SQL statement text.
- Create a statement and prepare the statement with the specified SQL statement text.

API

- SQLAllocStmt

Exceptions

- IC4SQLException

Public Members

bindParameter

Binds a user variable to a statement. This associates a variable with a parameter marker (indicated by '?') in an SQL statement text. When the statement is later executed, the value of the bound variable is used as the parameter value. This function is also used to specify any required data conversion. If another user variable was bound to this parameter prior to this function, that binding is removed. Overloaded versions of this function are provided to make calling this function easier.

This function provides ODBC version 1.0 support.

```
virtual RETCODE bindParameter(short pnumber,
                              short cType,
                              short sqlType,
                              unsigned long precision,
                              short scale,
                              void* value,
                              unsigned long* valueLength);
```

The parameters are the following:

pnumber	(in) Number of parameter marker being bound.
cType	(in) C language datatype of the variable.
sqlType	(in) SQL datatype to which the variable should be converted.
precision	(in) The precision of the parameter value being bound.
scale	(in) For numeric columns, the maximum number of decimal positions.
value	(in-deferred) The variable that will contain the value of the parameter at execution time.
valueLength	(in-deferred) At execution time, the actual length of the value being provided as the parameter. • If `NULL`, a null-terminated string is assumed. • If `SQL_NULL_DATA`, a `NULL` parameter value is assumed. • If `SQL_DATA_AT_EXEC`, it is assumed that the parameter value will be provided at execution time

Binds a user variable to a statement. This associates a variable with a parameter marker (indicated by '?') in an SQL statement. When the statement is executed, the value of the bound variable is used as the parameter value. This member function can also be used to specify any required data conversion. If another variable was bound to this parameter prior to this member function call, that binding is removed. Several overloaded versions of this member function are provided.

```
virtual RETCODE bindParameter(short pnumber,
                              short parameterType,
                              short cType,
                              short sqlType,
                              unsigned long precision,
                              short scale,
                              void* value,
                              long maxSize);
                              unsigned long* valueLength);
```

The parameters are the following:

pnumber	(in) Number of parameter marker being bound.
cType	(in) C language datatype of the variable.
sqlType	(in) SQL datatype to which the variable should be converted.
precision	(in) The precision of the parameter value being bound.
scale	(in) For numeric columns, the maximum number of decimal positions.
value	(in-deferred) The variable that will contain the value of the parameter at execution time.
valueLength	(in-deferred) At execution time, the actual length of the value being provided as the parameter. • If `NULL`, a null-terminated string is assumed. • If `SQL_NULL_DATA`, a `NULL` parameter value is assumed. • If `SQL_DATA_AT_EXEC`, it is assumed that the parameter value will be provided at execution time

API used: SQLSetParam

The following version uses the ODBC version 2.0 API SQLBindParameter. Your program should check whether SQLBindParameter is supported by the ODBC driver prior to invoking this function. SQLBindParameter provides the same functionality as SQLSetParam with the addition of two parameters to specify the type of parameter being bound and the maximum length that value might take at execution time. This member function supports binding of input/output and output parameters.

```
virtual RETCODE bindParameter(short pnumber,
                              short parameterType,
                              short cType,
                              short sqlType,
                              unsigned long precision,
                              short scale,
                              void* value,
                              long maxSize,
                              unsigned long* valueLength);
```

The parameters are the following:

pnumber	(in) Number of parameter marker being bound.

parameterType	(in) Type of the parameter being bound. Possible values are: • `SQL_PARAM_INPUT` • `SQL_PARAM_OUTPUT` • `SQL_PARAM_INPUT_OUTPUT`
cType	(in) C language datatype of the variable
sqlType	(in) SQL datatype to which the variable should be converted
precision	(in) The precision of the parameter value being bound
scale	(in) For numeric parameters, the maximum number of decimal positions
value	(in-deferred/out-deferred) The variable that, at execution time, will contain the input value of the parameter, the output value of the parameter, or both, depending on the parameter type
maxSize	(in) Maximum length that the parameter value being bound will be at execution time
valueLength	(in-deferred/out-deferred) At execution time, the actual length of the value being provided as the parameter. • If `NULL`, a null-terminated string is assumed. • If `SQL_NULL_DATA`, a `NULL` parameter value is assumed. • If `SQL_DATA_AT_EXEC`, it is assumed that the parameter value will be provided at execution time When the parameter is an output or input/output parameter, this location returns the amount of data available to return or a special value defined by ODBC.

API used: SQLBindParameter (if ODBC driver supported ODBC version 2.0)

To bind a user variable of a particular type to a statement, use one of the following versions.

```
virtual RETCODE bindParameter(short pnumber,
                              short& value,
                              short sqlType =SQL_SMALLINT);
virtual RETCODE bindParameter(short pnumber,
                              long& value,
                              short sqlType =SQL_INTEGER);
virtual RETCODE bindParameter(short pnumber,
                              float& value,
                              short sqlType =SQL_REAL);
virtual RETCODE bindParameter(short pnumber,
                              double& value,
                              short sqlType =SQL_DOUBLE);
virtual RETCODE bindParameter(short pnumber,
                              DATE_STRUCT& value,
                              short sqlType =SQL_DATE);
virtual RETCODE bindParameter(short pnumber,
                              TIME_STRUCT& value,
                              short sqlType =SQL_TIME);
virtual RETCODE bindParameter(short pnumber,
                              TIMESTAMP_STRUCT& value,
                              short sqlType =SQL_TIMESTAMP);
```

The parameters are the following:

pnumber	(in) Number of parameter marker being bound.
value	(in) The variable being bound.
sqlType	(in) SQL datatype to which the variable should be converted.

API used: SQLSetParam (ODBC version 1.0)

To bind a char* user variable to a statement, use this version:

```
virtual RETCODE bindParameter(short pnumber,
                              unsigned long precision,
                              const char* value,
                              unsigned long* length =NULL,
                              short sqlType =SQL_CHAR);
```

The parameters are the following:

pnumber	(in) Number of parameter marker being bound.
precision	(in-deferred) The precision of the parameter value being bound.
value	(in-deferred) The variable being bound.
length	(in-deferred) At execution time, the actual length of the value being provided as the parameter. • If NULL, a null-terminated string is assumed. • If SQL_NULL_DATA, a NULL parameter value is assumed. • If SQL_DATA_AT_EXEC, it is assumed that the parameter value will be provided at execution time.
sqlType	(in) SQL datatype to which the variable should be converted.

API used: SQLSetParam (ODBC version 1.0)

To bind a user variable to a statement using an IC4SQLVariable object, use the following version. This member function requires ODBC version 2.0.

```
virtual RETCODE bindParameter(const IC4SQLVariable& variable);
```

The parameters are the following:

variable	(in) An IC4SQLVariable object describing the variable to bind. The variable object must contain a non-zero position value indicating the parameter number to which the variable is to be bound. If the variable object was created using a constructor where the *length* parameter was specified, the variable can be used only if the ODBC driver being used supports SQLBindParameter.

API used: SQLBindParameter or SQLSetParam

Return Values

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Exceptions

- IC4SQLException

bindParameters

```
virtual RETCODE bindParameters(const IC4SQLVariableList& varlist);
```

Binds multiple user variables to a statement using an `IC4SQLVariableList` object.

The parameters are the following:

varlist	(in) Variable list containing variables to bind. Each variable object in the list that contains a non-zero position value is bound to the parameter with the corresponding parameter marker. Otherwise, the next parameter marker number following the last one bound from this list is used. If the first variable in the list has a position number of zero, then 1 is used.

API used: SQLSetParam

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

cancel

```
virtual RETCODE cancel();
```

Cancel an outstanding `SQL_NEED_DATA` situation, or cancel processing on this statement object in another thread. If the server supports asynchronous processing, this function can cancel processing of an asynchronously submitted statement.

API used: SQLCancel

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

execute

```
virtual RETCODE execute();
virtual RETCODE execute(IC4SQLResultSet& results);
```

Executes a statement that was prepared previously. For a SELECT statement, the version that returns a result set must be used. If the statement is not a SELECT statement, the result set is closed.

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results of a SELECT statement.

API used: SQLExecute

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NEED_DATA`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

executeDirect

Dynamically prepares and executes the specified SQL statement text.

Use the following version to execute a statement that is not a SELECT statement:

```
virtual RETCODE executeDirect(const char* statement,
                              long length =SQL_NTS);
```

Use the following version to execute a SELECT statement. If the statement is not a SELECT statement, the result set is closed.

```
virtual RETCODE executeDirect(IC4SQLResultSet& results,
                              const char* statement,
                              long length = SQL_NTS);
```

The parameters are the following:

results	(out) The `IC4SQLResultSet` object to receive the results of a SELECT statement.
statement	(in) The SQL statement text.
length	(in) The length of the statement text.

API used: SQLExecDirect

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NEED_DATA`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

getParameterDescription

```
virtual RETCODE getParameterDescription(unsigned short markerNumber,
                                        short& sqlType,
                                        unsigned long& precision,
                                        short& scale,
                                        short& nullable);
```

Retrieves attributes of a variable bound to a parameter marker. This returns the parameter values that were specified when `bindParameter()` was done.

The parameters are the following:

markerNumber	(in) Parameter marker number.
sqlType	(out) SQL datatype to which the user value is converted.
precision	(out) The precision of the parameter.
scale	(out) For numeric parameters, the maximum number of decimal positions.
nullable	(out) Whether `NULL`s are allowed for this field. Possible values are: • `SQL_NO_NULLS` • `SQL_NULLABLE` • `SQL_NULLABLE_UNKNOWN`

API used: SQLDescribeParam

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

getStatusInformation

```
virtual RETCODE getStatusInformation(IString& sqlState,
                                     long& nativeError,
                                     IString& messageText);
virtual RETCODE getStatusInformation(char* sqlState,
                                     long& nativeError,
                                     char* messageText,
                                     short messageTextLength,
                                     short& messageTextBytesAvailable);
```

Returns status information associated with the last operation performed if the last operation returned `SQL_SUCCESS_WITH_INFO`. If another operation is performed with this object, the information for the previous operation can no longer be accessed. Once the information is retrieved, attempting to retrieve it again returns `SQL_NO_DATA_FOUND`. If the last operation returned `SQL_SUCCESS` or threw an exception, `SQL_NO_DATA_FOUND` is returned.

The parameters are the following:

sqlState	(out) User-allocated storage to receive the value of SQLSTATE. Must be at least 5 bytes long.
nativeError	(out) User-allocated storage to receive the native error value.
messageText	(out) User-allocated storage to receive the error text.
messageTextLength	(in) Length of messageText.
messageTextBytesAvailable	(out) Length of text available to return.

API used: SQLError

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NO_DATA_FOUND`

Exceptions

- IC4SQLException

moreResults

```
virtual RETCODE moreResults();
```

Determines whether more counts are available after batched update, insert, or delete operations. The counts tell how many rows were affected by the operation.

API used: SQLMoreResults

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

numberParameters

```
virtual RETCODE numberParameters(unsigned short& number);
```

Returns the number of parameters in a statement.

The parameters are the following:

number	(out) The number of parameters.

API used: SQLNumParams

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

parameterData

```
virtual RETCODE parameterData(void* * value);
```

Use this member function with `putData()` to supply parameter data at statement execution time. It returns the value that is used to find the data that will be provided as a parameter on an `execute()`.

The parameters are the following:

value	(out) Pointer to the pointer to the user buffer specified on `setParameter()`.

API used: SQLParamData

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NEED_DATA`

Exceptions

- IC4SQLException

prepare

```
virtual RETCODE prepare(const char* statement,
                        long length =SQL_NTS);
```

Prepares a statement.

The parameters are the following:

statement	(in) SQL statement text.
length	(in) Length of statement text. The default value indicates that the string is null terminated.

API used: SQLPrepare

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

putData

```
virtual RETCODE putData(const void* value,
                        long int length);
```

Use this member function with `parameterData()` to supply parameter data at statement execution time.

The parameters are the following:

value	(in) Parameter data being provided.
length	(in) Length of parameter data.

API used: SQLPutData

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

Exceptions

- IC4SQLException

rowsAffected

```
virtual RETCODE rowsAffected(long& rows);
```

Returns the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE SQL statement executed against a table.

The parameter is the following:

rows	(out) The number of rows in a table that were affected. 0 is returned if the previous operation was not an UPDATE, INSERT, or DELETE.

API used: SQLRowCount

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

setCursorName

```
virtual RETCODE setCursorName(const char* cursorName,
                              short length =SQL_NTS);
```

Sets a cursor name to be associated with the next result set that is generated. If the application does not set a cursor name, the database generates a name. If the application sets a cursor name and generates a result set, and then this statement object is used to generate another result set before closing the first result set, a different cursor name must be set; otherwise an exception is thrown.

The parameters are the following:

cursorName	(in) The cursor name
length	(in) Length of cursor name. The default value indicates that the string is null terminated.

API used: SQLSetCursorName

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

setParameterOptions

```
virtual RETCODE setParameterOptions(unsigned long numberOfValues,
                                    unsigned long* currentIndex);
```

Specifies number of values being provided when using multiple sets of values for bound parameters. This allows bulk insert operations, for example. Specifying multiple sets of parameters requires that arrays of values be provided at the locations specified with `bindParameter()`. It is not possible to specify these arrays if the version of `bindParameter()` that takes `IC4SQLVariable` or `IC4SQLVariableList` was used for any parameter marker.

The parameters are the following:

numberOfValues	(in) Number of values for each parameter.
currentIndex	(out-deferred) Location where the current row number of data being executed is returned during batch execution of a statement. If a failure occurs, the application can determine which set of data caused the problem.

API used: SQLParamOptions

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

setStatementOption

```
virtual RETCODE setStatementOption(unsigned short option,
                                   signed long value);
```

Sets a statement option.

The parameters are the following:

option	(in) The option that is to be set.
value	(in) Value of option.

API used: SQLSetStmtOption

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

statementOption

```
virtual RETCODE statementOption(unsigned short option,
                                unsigned long& value);
```

Get the current value of a statement option.

The parameters are the following:

option	(in) The option for which the value is to be returned.
value	(out) Value of option.

API used: SQLGetStmtOption

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

unbindParameters

```
virtual RETCODE unbindParameters();
```

Unbinds all parameters previously set for this statement.

API used: SQLFreeStmt

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

unprepare

```
virtual RETCODE unprepare();
```

Puts a statement back into an unprepared state, freeing up AS/400 resources. All other attributes of the statement remain unchanged.

API used: SQLFreeStmt

Return Values

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

Exceptions

- IC4SQLException

IC4SQLVariable

The `IC4SQLVariable` class represents the information needed to bind a column, set a variable, and get column data after a row is fetched. Objects of this class can be used with an `IC4SQLVariableList` object to simplify passing a set of variables as a single parameter to certain SQL member functions.

Derivation

`IC4SQLVariable` does not derive from another class.

Header File

`ic4sqva.hpp`

IC4SQLVariable

Constructors

The following versions are provided to allow defaulting parameters. If *position* is allowed to default to 0 and the variable is added to a variable list object, the position is set to the next position in the list following the previously added variable. If not added to a variable list, the position must be explicitly set before using the variable object to bind a parameter or column.

```
IC4SQLVariable();
IC4SQLVariable(const IC4SQLVariable& variable);
IC4SQLVariable(const IC4SQLVariable& variable,
               short position);

IC4SQLVariable(void* variable,
               short position,
               short parameterType,
               short cType,
               short sqlType,
               unsigned long precision,
               short scale,
               long length =SQL_NTS,
               unsigned long* bytesAvailable =NULL);
```

These constructors allow requesting data conversions that are different from the defaults. For example, conversion from character string to numeric, or to specify a different length or precision.

```
IC4SQLVariable(short& variable,
               short position =0,
               short sqlType =SQL_SMALLINT,
               long* bytesAvailable =NULL);
IC4SQLVariable(long& variable,
               short position =0,
               short sqlType =SQL_INTEGER,
               long* bytesAvailable =NULL);
IC4SQLVariable(float& variable,
               short position =0,
               short sqlType =SQL_REAL,
               long* bytesAvailable =NULL);
IC4SQLVariable(double& variable,
               short position =0,
               short sqlType =SQL_DOUBLE,
               long* bytesAvailable =NULL);
IC4SQLVariable(char* variable,
               unsigned long precision =0,
               short position =0,
               short sqlType = SQL_CHAR,
               long* bytesAvailable = NULL);
IC4SQLVariable(DATE_STRUCT& variable,
               short position =0,
               short sqlType =SQL_DATE,
               long* bytesAvailable =NULL);
IC4SQLVariable(TIME_STRUCT& variable,
               short position =0,
               short sqlType =SQL_TIME,
               long* bytesAvailable =NULL);
IC4SQLVariable(TIMESTAMP_STRUCT& variable,
               short position =0,
               short sqlType =SQL_TIMESTAMP,
               long* bytesAvailable =NULL);
```

The parameters are the following:

variable	(in) The user variable to be bound to a parameter marker or to a column in a result set. Your program passes the variable to `bindParameter()` or `bindColumn()` later. The following parameters are ignored when binding a column: • *parameterType* • *sqlType* • *precision* • *scale*
position	(in) The position of the parameter or column to bind.
parameterType	(in) The parameter type. Possible values are: • `SQL_PARAM_INPUT` • `SQL_PARAM_OUTPUT` • `SQL_PARAM_INPUT_OUTPUT`
cType	(in) The C language type is determined by the type of the user variable provided.
sqlType	(in) The SQL type of the parameter or column with which this user variable is to be associated.
precision	(in) The precision of the variable.
scale	(in) The maximum number of decimal positions for the variable.
length	(in) When used to bind a column, this is the length of the column variable. When used to bind a parameter, this is the actual length of the parameter variable or SQL_NTS if the string is null-terminated.
bytesAvailable	(in) Pointer that is later passed to `bindParameter()` or `bindColumn()` to specify a deferred output location.

Public Members

assignment operator

```
IC4SQLVariable& operator=(const IC4SQLVariable& var);
```

Refers to a variable object.

cType

```
short cType() const;
```

Returns the value of the *cType* data member.

longMaxSize

`**longMaxSize**() const;`

Returns the value of the maxSize() data member.

maxSize

`long **maxSize**() const;`

Returns the maximum size in bytes of the variable. When targeting OS/400, this member function has no meaning.

parameterType

`**parameterType**() const;`

Returns the value of the parameterType() data member.

position

`short **position**() const;`

Returns value of the *position* data member.

precision

`unsigned long **precision**() const;`

Returns the value of the *precision* data member.

returnSize

`unsigned long* **returnSize**() const;`

Returns the value of the *returnSize* data member.

scale

`short **scale**() const;`

Returns the value of the *scale* data member.

setCType

`IC4SQLVariable& **setCType**(short *x*);`

Sets the value of the *cType* data member.

setMaxSize

`IC4SQLVariable& **setMaxSize**(long x);`

Sets the maximum size in bytes of the variable. When targeting OS/400, this member function has no affect on the object.

setParameterType

```
SetParameterType(short x);
```

Sets the value of the parameterType() data member.

setPosition

```
IC4SQLVariable& setPosition(short x);
```

Sets the value of the *position* data member.

setPrecision

```
IC4SQLVariable& setPrecision(unsigned long x);
```

Sets the value of the *precision* data member.

setReturnSize

```
IC4SQLVariable& setReturnSize(unsigned long* x);
```

Sets the value of the *returnSize* data member.

setScale

```
IC4SQLVariable& setScale(short x);
```

Sets the value of the *scale* data member.

setSqlType

```
IC4SQLVariable& setSqlType(short x);
```

Sets the value of the *sqlType* data member.

setVariable

```
IC4SQLVariable& setVariable(void* x);
```

Sets the value of the *variable* data member.

sqlType

```
short sqlType() const;
```

Returns the value of the *sqlType* data member.

variable

```
void* variable() const;
```

Returns the value of the *variable* data member.

IC4SQLVariableList

The `IC4SQLVariableList` class represents a collection of `IC4SQLVariable` objects that can be passed as a single parameter to certain database member functions. A variable list can be used with `IC4SQLStatement::bindParameters()`, `IC4SQLResultSet::bindColumns()`, and `IC4SQLResultSet::getData()`.

Derivation

`IC4SQLVariableList` does not derive from another class.

Header File

`ic4sqvl.hpp`

Constructors

```
IC4SQLVariableList();
IC4SQLVariableList(const IC4SQLVariableList& list);
```

Public Members

add

```
IC4SQLVariableList& add(const IC4SQLVariable& variable);
```

Adds a variable to the list. The variable is placed at the position specified by the `position` attribute of the variable if it is greater than 0. Otherwise, the variable is placed in the list at the position following the most recently added variable.

The parameter is the following:

variable	The `IC4SQLVariable` object that is to be added to the list.

numberOfElements

```
virtual unsigned short numberOfElements();
```

Returns the number of elements in a variable list.

operator=

```
IC4SQLVariableList& operator=(const IC4SQLVariableList& list);
```

Assignment operator.

The parameter is the following:

list	The `IC4SQLVariableList` object that is to be copied.

operator<<

```
IC4SQLVariableList& operator<<(const IC4SQLVariable& variable);
```

Insertion operator: adds a variable to a variable list.

The parameter is the following:

variable	The `IC4SQLVariable` object that is to be added to the list.

removeAll

```
void removeAll();
```

Removes all variables in a variable list.

removeAt

```
IC4SQLVariableList& removeAt(unsigned short position);
```

Removes a variable from a variable list.

The parameters are the following:

position	(in) The 1-based position of the variable to remove.

Cursor

The `IC4SQLVariableList` class has a nested class, `Cursor`, to allow iterating through a variable list using cursor member functions identical to those provided by the Collection Class Library.

The member functions available are the following:

```
Cursor(const IC4SQLVariableList& list);
IBoolean setToFirst();
IBoolean setToNext();
IBoolean setToPrevious();
IBoolean setToLast();
IBoolean isValid() const;
void invalidate();
const IC4SQLVariable& element() const;
IBoolean operator==(const Cursor& cursor) const;
```

IC4SQLVariableList

Chapter 4. Data Queue Classes

Data queues exist in a library on the AS/400 and are a simple and commonly used mechanism for communication between multiple tasks or programs. They are often used for communication between a client workstation and a server job.

IC4BaseDataQueue

The `IC4BaseDataQueue` class is an abstract base class representing common functions for all data queues on an AS/400. It provides a common interface to both keyed and nonkeyed data queues.

Derivation

`IC4BaseDataQueue` does not derive from another class.

Header File

`ic4basdq.hpp`

Constructors

`IC4BaseDataQueue` is an abstract base class. Constructing an instance is not allowed.

Public Members

asyncRead

```
virtual asyncHandle asyncRead(char* readBuffer,
                              unsigned long readBufferLength,
                              unsigned long& readLength,
                              long waitTime =0,
                              char* senderInformation =NULL) =0;
```

Submits a request to read an entry from the data queue. Use `checkData()` to access the entry.

The parameters are the following:

readBuffer	(out-deferred) User-allocated storage to receive the entry which is read.
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
readLength	(out-deferred) The available length of the entry that was read.
waitTime	(in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue.

senderInformation	(out-deferred) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.

cancelRequest

```
virtual void cancelRequest(asyncHandle readHandle ) =0;
```

Cancels a pending `asyncRead()` call. The value for *readHandle* was returned by `asyncRead()`.

checkData

```
virtual void checkData(asyncHandle readHandle ) =0;
```

Retrieves any data that was read using `asyncRead`. The value for *readHandle* was returned by `asyncRead()`. If an entry was read, the entry and the length of the entry are returned in the variables specified on `asyncRead()`. If *senderInformation* was requested, it is returned.

clear

```
virtual void clear();
```

Removes all entries from the data queue.

Exceptions

- IC4DataQueueException

close

```
virtual void close();
```

Closes the communications conversation that was started by `open()`.

Exceptions

- IC4DataQueueException

convert

```
virtual IBoolean convert() const;
```

Returns True if the entry is converted between local system data format and EBCDIC when reading or writing an entry.

For Windows 95 and for Windows NT, the local system format is ASCII, and the target (OS/400) format is EBCDIC.

dataLength

```
virtual unsigned long dataLength() const;
```

Returns the available length of the last entry accessed using read or peek.

destroy

```
virtual void destroy();
```

Deletes the data queue from the AS/400 and does an implicit `close()`.

Exceptions

- IC4DataQueueException

force

```
virtual Force force() const;
```

Returns *Force_Yes* if the `FORCE` attribute was set when the data queue was created. This attribute specifies whether the data queue is forced to auxiliary storage when entries are sent or received. Other possible values are *Force_No*.

Exceptions

- IC4DataQueueException

isKeyed

```
virtual IBoolean isKeyed() const =0;
```

Returns True if the data queue is keyed.

isOpen

```
IBoolean isOpen() const;
```

Returns True if the data queue is open.

maxRecordLength

```
virtual unsigned long maxRecordLength() const;
```

Returns the maximum length of an entry that can be put on the data queue.

Exceptions

- IC4DataQueueException

name

```
virtual inline IString name() const;
```

Returns the fully qualified name of the data queue specified on the constructor.

open

```
virtual void open();
```

Starts a conversation with the system specified on the constructor and verifies that the data queue exists on the AS/400 system.

Exceptions

- IC4DataQueueException

operator==

```
virtual IBoolean operator==(const IC4BaseDataQueue& dataQueue) const;
```

Equality operator: returns True if the system names and the qualified data queue names are the same.

Exceptions

- IC4DataQueueException

operator!=

```
virtual IBoolean operator!=(const IC4BaseDataQueue& dataQueue) const;
```

Inequality operator: returns True if the system names or the qualified data queue names are different.

Exceptions

- IC4DataQueueException

operator<

```
virtual IBoolean operator<(const IC4BaseDataQueue& dataQueue) const;
```

Less-than operator: returns True if the left operand is less than the right operand as determined by the alphabetical order of the system name and then the qualified data queue name.

Exceptions

- IC4DataQueueException

operator>

```
virtual IBoolean operator>(const IC4BaseDataQueue& dataQueue) const;
```

Greater-than operator: returns True if the left operand is greater than the right operand as determined by the alphabetical order of the system name and then the qualified data queue name.

Exceptions

- IC4DataQueueException

operator<<

```
virtual IC4BaseDataQueue& operator<<(const char* writeString) =0;
```

Insertion operator: writes an entry to the data queue. *writeString* must be null-terminated.

operator>>

```
virtual IC4BaseDataQueue& operator>>(IString& readString) =0;
```

Extraction operator: reads an entry from the data queue and places it in *readString*.

order

```
virtual Order order() const;
```

Returns the value of the attribute that specifies the sequence in which entries are received from the data queue. This applies to `asyncRead()`, `read()` and `peek()` operations. For information on *order*, see “Order” on page 85.

Exceptions

- IC4DataQueueException

peek

```
virtual void peek(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL) =0;
virtual void peek(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL) =0;
```

Reads an entry from the data queue without removing the entry from the queue.

The parameters are the following:

readString	(out) String to receive the entry.
readBuffer	(out) User-allocated storage to receive the entry.
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
waitTime	(in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue.
senderInformation	(out) User-allocated storage to receive sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.

read

```
virtual void read(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL) =0;
virtual void read(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL) =0;
```

Reads an entry from the data queue.

The parameters are the following:

readString	(out) String to receive the entry.
readBuffer	(out) User-allocated storage to receive the entry.
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
waitTime	(in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue.
senderInformation	(out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.

senderID

```
virtual IBoolean senderID() const;
```

Returns True if sender information is saved with this data queue.

Exceptions

- IC4DataQueueException

setConvert

```
virtual void setConvert(IBoolean value);
```

Sets whether the entry is converted between local system data format and EBCDIC when reading or writing an entry.

For Windows 95 and for Windows NT, the local system format is ASCII, and the target (OS/400) format is EBCDIC.

Exceptions

- IC4DataQueueException

systemName

```
virtual IString systemName() const;
```

Returns the name of the AS/400 system that the data queue is located on.

Exceptions

- IC4DataQueueException

textDescription

```
virtual IString textDescription() const;
```

Returns the text that describes this data queue.

Exceptions

- IC4DataQueueException

write

```
virtual void write(const char* writeString,
                   unsigned long writeStringLength =0,
                   IBoolean commit =True) =0;
```

Writes an entry to the data queue.

The parameters are the following:

writeString	(in) Entry to be put on the data queue.
writeStringLength	(in) Size of the writeString. If this value is 0, *writeString* must be null-terminated.
commit	(in) True indicates the data is to be committed on the write before this function returns. This is an optional parameter. If not specified, the data is committed.

Enumerations

The following describes the Force enumeration and the Order enumeration.

Force

```
enum Force { Force_No=0, Force_Yes };
```

Use these values for the `FORCE` data queue attribute. The attribute specifies whether the data queue is forced to auxiliary storage when entries are written to or read from the data queue.

Order

```
enum Order { Order_LIFO=0, Order_FIFO };
```

Use these values for the `SEQ` data queue attribute. This attribute specifies the sequence in which entries are received from the data queue.

IC4DataQueue

The `IC4DataQueue` class represents an AS/400 data queue that is accessed sequentially rather than with a key.

Derivation

```
IC4BaseDataQueue
 IC4DataQueue
```

Header File

`ic4dq.hpp`

Constructors

```
IC4DataQueue(const char* dataQueueName,
             const char* systemName =NULL);
```

```
IC4DataQueue(const IC4DataQueue& dataQueue);
```

The parameters are the following:

dataQueueName	(in) The qualified name of the data queue, specified as a path name in the library file system. For example, `/QSYS.LIB/MYLIB.LIB/MYDATAQ.DTAQ`. The library can be one of the following: • `a library name` • `%CURLIB%` • `%LIBL%`
systemName	(in) The name of the AS/400 system that the data queue is located on. This is an optional parameter. If not specified, the default system name for Client Access is used.
dataQueue	(in) An `IC4DataQueue` object. If this object is open, then the newly constructed object is opened.

Exceptions

- IC4DataQueueException

Public Members

asyncRead

```
virtual asyncHandle asyncRead(char* readBuffer,
                              unsigned long readBufferLength,
                              unsigned long& readLength,
                              long waitTime =0,
                              char* senderInformation =NULL);
```

Submits a request to read an entry from the data queue. Use `checkData()` to access the entry.

The parameters are the following:

readBuffer	(out-deferred) String to receive the entry
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
readLength	(out-deferred) The available length of the entry that was read.
waitTime	(in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue.
senderInformation	(out-deferred) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.

Exceptions

- IC4DataQueueException

cancelRequest

```
virtual void cancelRequest(asyncHandle readHandle);
```

Cancels a pending `asyncRead()`. The value for *readHandle* was returned by `asyncRead()`.

Exceptions

- IC4DataQueueException

checkData

```
virtual void checkData(asyncHandle readHandle);
```

Retrieves any data that was read by `asyncRead()`. The value for *readHandle* was returned by `asyncRead()`. If an entry was read, the entry and the length of the entry are returned in the variables specified on `asyncRead()`. Sender information is returned, if it was requested.

Exceptions

- IC4DataQueueException

create

```
virtual void create(unsigned long maxRecordLength =80,
                    IC4BaseDataQueue::Order order =Order_FIFO,
                    IC4BaseDataQueue::Force force =Force_No,
                    const char* authority =NULL,
                    IBoolean senderID =False,
                    const char* textDescription =NULL);
```

IC4DataQueue

Creates a non-keyed data queue on the AS/400 and does an implicit `open()`. The data queue can be created in a specific library or in the current library.

The parameters are the following:

maxRecordLength	(in) The maximum length of an entry that can be put on the data queue. This is an optional parameter. The default value is 80 bytes. The allowable values are 1-64512.
order	(in) The sequence in which entries are received from the data queue. This is an optional parameter. For information on *order*, see "Order" on page 85.
force	(in) Specifies if the data queue should be forced to auxiliary storage when entries are sent or received. This is an optional parameter. For information on *force*, see "Force" on page 85.
authority	(in) The public access authority given to users who do not have specific authority to the data queue. This is an optional parameter which defaults to `*LIBCRTAUT`. Other possible values are: • `*CHANGE` • `*ALL` • `*USE` • `*EXCLUDE` • `authorization list name`
senderID	(in) True indicates sender information should be saved when entries are put on the data queue. This is an optional parameter. The default is not to save sender information.
textDescription	(in) Description of the data queue. This is an optional parameter and the default is a blank string.

Exceptions

- IC4DataQueueException

isKeyed

```
virtual inline IBoolean isKeyed() const;
```

Returns false because this class represents a non-keyed queue.

operator=

```
IC4DataQueue& operator=(const IC4DataQueue& dataQueue);
```

Assignment operator: If the data queue object that is being copied is open, then this object is open when the assignment operation is complete.

Exceptions

- IC4DataQueueException

operator<<

```
virtual IC4BaseDataQueue& operator<<(const char* writeString);
```

Insertion operator: writes an entry to the data queue. The string to be written must be null terminated.

Exceptions

- IC4DataQueueException

operator>>

```
virtual IC4BaseDataQueue& operator>>(IString& readString);
```

Extraction operator: reads an entry from the data queue and places it in *readString*.

Exceptions

- IC4DataQueueException

peek

```
virtual void peek(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL);
virtual void peek(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL);
```

Reads an entry from the data queue without removing the entry from the queue. A subsequent `read()` or `peek()` will return the same entry.

The parameters are the following:

readString	(out) String to receive the entry.
readBuffer	(out) User-allocated storage to receive the entry.
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
waitTime	(in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue.
senderInformation	(out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.

Exceptions

- IC4DataQueueException

read

```
virtual void read(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL);
virtual void read(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL);
```

Reads an entry from the data queue.

The parameters are the following:

readString	(out) String to receive the entry.
readBuffer	(out) User-allocated storage to receive the entry.
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
waitTime	(in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue.
senderInformation	(out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.

Exceptions

- IC4DataQueueException

write

```
virtual void write(const char* writeString,
                   unsigned long writeStringLength =0,
                   IBoolean commit =True);
```

Writes an entry to the data queue.

The parameters are the following:

writeString	(in) Entry to be placed on the data queue.
writeStringLength	(in) Size of the writestring. If this value is 0, the *writestring* must be null-terminated.
commit	(in) True indicates the data is to be committed on the write before this function returns. This is an optional parameter. If not specified, the data is committed.

Exceptions

- IC4DataQueueException

Inherited Public Members

Member	Class	Page
clear	IC4BaseDataQueue	80
close	IC4BaseDataQueue	80
convert	IC4BaseDataQueue	80
dataLength	IC4BaseDataQueue	80
destroy	IC4BaseDataQueue	81
force	IC4BaseDataQueue	81
isOpen	IC4BaseDataQueue	81
maxRecordLength	IC4BaseDataQueue	81
name	IC4BaseDataQueue	81
open	IC4BaseDataQueue	81
operator==	IC4BaseDataQueue	82
operator!=	IC4BaseDataQueue	82
operator<	IC4BaseDataQueue	82
operator>	IC4BaseDataQueue	82
order	IC4BaseDataQueue	83
senderID	IC4BaseDataQueue	84
setConvert	IC4BaseDataQueue	84
systemName	IC4BaseDataQueue	84
textDescription	IC4BaseDataQueue	85

IC4KeyedDataQueue

The `IC4KeyedDataQueue` class represents an AS/400 data queue that is accessed using a key value. The use of a key value allows data queue entries to be processed selectively. For example, a program can access just the entries with a certain key, ignoring other entries on the data queue.

Derivation

```
IC4BaseDataQueue
 IC4KeyedDataQueue
```

Header File

`ic4keydq.hpp`

Constructors

```
IC4KeyedDataQueue(const char* dataQueueName,
                  const char* systemName =NULL,
                  const char* key =NULL,
                  unsigned short keyLength =0);
```

```
IC4KeyedDataQueue(const IC4KeyedDataQueue& dataQueue);
```

The parameters are the following:

dataQueueName	(in) The qualified name of the data queue specified as a path name in the library file system. For example, `/QSYS.LIB/MYLIB.LIB/MYDATAQ.DTAQ`. The library can be one of the following: • `a library name` • `%CURLIB%` • `%LIBL%`
systemName	(in) The name of the AS/400 system that the data queue is located on. This is an optional parameter. If not specified, the default system name for Client Access is used.
key	(in) The default key to be used for operations on this queue. This is an optional parameter. If not specified, the key must be set later or passed on member function calls. Using this parameter has the same effect as using `setKey()`.
keyLength	(in) Size of the key. If this value is 0, the key value must be null-terminated. The key will be padded with blanks or truncated if it is not the same length as the value specified when the data queue was created.
dataQueue	(in) An `IC4KeyedDataQueue` object. If this object is open, then the newly constructed object is opened.

Exceptions

- IC4DataQueueException

Public Members

asyncKeyReturned

```
virtual IString asyncKeyReturned() const;
```

Returns the key for the entry returned by the last successful `checkData()` operation.

Exceptions

- IC4DataQueueException

asyncRead

```
virtual asyncHandle asyncRead(const char* key,
                              char* readBuffer,
                              unsigned long readBufferLength,
                              unsigned long& readLength,
                              unsigned short keyLength =0,
                              long waitTime =0,
                              char* senderInformation =NULL,
                              SearchType searchType =Search_Null);
virtual asyncHandle asyncRead(char* readBuffer,
                              unsigned long readBufferLength,
                              unsigned long& readLength,
                              long waitTime =0,
                              char* senderInformation =NULL);
```

Submits a request to read an entry from the data queue. Use `checkData()` to access the entry. The *key* and *searchType* can be passed explicitly, or a default value can be used. If a key is not passed, the key specified on the last `setKey()` is used to select the entry. If the key value was never set, a blank key is used.

The parameters are the following:

key	(in) Key value to use for comparison when selecting an entry.
readBuffer	(out-deferred) User-allocated storage to receive the entry.
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
readLength	(out-deferred) The available length of the entry that was read.
keyLength	(in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created.
waitTime	(in) Number of seconds to wait if no entries on the queue match the key. This is an optional parameter. The default is not to wait. A value of -1 indicates to wait until an there is a matching entry on the queue.
senderInformation	(out-deferred) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.
searchType	(in) Comparison value to use when searching for a specific key. This is an optional parameter. If `Search_Null` is passed, the value from the last `setSearchType()` is used. If `setSearchType()` has not been done, *Search_EQ* is used.

Exceptions

- IC4DataQueueException

cancelRequest

```
virtual void cancelRequest(asyncHandle readHandle);
```

Cancels a pending `asyncRead()`. The value for *readHandle* was returned by `asyncRead()`.

Exceptions

- IC4DataQueueException

checkData

```
virtual void checkData(asyncHandle readHandle);
```

Retrieves the entry that was read using `asyncRead()`. If an entry matches the key value, the entry and the length of the entry are returned in the variables specified on `asyncRead()`. Sender information is returned if it was requested.

Exceptions

- IC4DataQueueException

clearByKey

```
virtual void clearByKey();
virtual void clearByKey(const char* key,
                        unsigned short keyLength =0);
```

Removes all entries from the data queue that match a key. The key can be passed explicitly, or a default value can be used. If a key is not passed, the key specified on the last `setKey()` is used to select the entries. If the key value was never set, a blank key is used.

The parameters are the following:

key	(in) String that contains the key value to use when removing entries.
keyLength	(in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created.

Exceptions

- IC4DataQueueException

convertKey

```
IBoolean convertKey() const;
```

Returns True if the key is converted between local system data format and EBCDIC when reading or writing an entry.

create

```
virtual void create(unsigned long maxRecordLength =80,
                    unsigned short keyLength =10,
                    IC4BaseDataQueue::Force force =Force_No,
                    const char* authority =NULL,
                    IBoolean senderID =False,
                    const char* textDescription =NULL);
```

Creates a data queue on the AS/400 and does an implicit `open()`. The data queue can be created in a specific library or in the current library.

The parameters are the following:

maxRecordLength	(in) The maximum length of an entry that can be put on the data queue. This is an optional parameter. The default value is 80 bytes. Allowable values are 1-64512.
keyLength	(in) The maximum allowable length of the key field. This is an optional parameter. The default value is 10 bytes. Allowable values are 1-256.
force	(in) Specifies if the data queue should be forced to auxiliary storage when entries are sent or received. This is an optional parameter. For information on *force*, see "Order" on page 85.
authority	(in) The public access authority given to users who do not have specific authority to the data queue. This is an optional parameter which defaults to `*LIBCRTAUT`. Other possible values are: • `*CHANGE` • `*ALL` • `*USE` • `*EXCLUDE` • `authorization list name`
senderID	(in) If true, sender information will be saved when entries are put on the data queue. This is an optional parameter. The default is not to save sender information.
textDescription	(in) Description of the data queue. This is an optional parameter. The default is a blank string.

Exceptions

- IC4DataQueueException

isKeyed

```
virtual inline IBoolean isKeyed() const;
```

Returns True because this class represents a keyed data queue.

key

```
virtual IString key() const;
```

Returns the key used for comparisons when selecting an entry. The key can be set on the constructor or using `setKey()`. If this member function is used before `open()`, the value returned matches the value specified by the last `setKey()` member function call. After `open()`, the value returned may be blank padded or truncated so that it is the same length as the value specified when the data queue was created.

keyLength

```
virtual unsigned short keyLength() const;
```

Returns the length of the key field for the data queue.

Exceptions

- IC4DataQueueException

keyReturned

```
virtual IString keyReturned() const;
```

Returns the key for the entry returned by the last successful `read()` or `peek()` operation.

Exceptions

- IC4DataQueueException

open

```
virtual void open();
```

Starts a conversation with the system specified on the constructor and verifies that the data queue exists on the AS/400 system.

Exceptions

- IC4DataQueueException

operator=

```
IC4KeyedDataQueue& operator=(const IC4KeyedDataQueue& dataQueue);
```

Assignment operator: If the KeyedDataQueue object that is being copied is open, then this object is open when the assignment operation is complete.

Exceptions

- IC4DataQueueException

operator<<

```
virtual IC4BaseDataQueue& operator<<(const char* writeString);
```

Insertion operator: writes an entry to the data queue. The data must be null-terminated. Entries are written using the key value specified on `setKey()`. If `setKey()` has not been done, a blank key is used.

Exceptions

- IC4DataQueueException

operator>>

```
virtual IC4BaseDataQueue& operator>>(IString& readString);
```

Extraction operator: removes an entry from the data queue and returns it in *readString*. Sender information is not returned and this member function does not wait for an entry to appear on the data queue.

The last value specified on `setSearchType()` is used when comparing keys. If `setSearchType()` has not been done, *Search_EQ* is used. The key specified on the last `setKey()` is used to select the entry. If the key value was never set, a blank key is used. If no entries match the key specified on `setKey`, an empty string is returned.

Exceptions

- IC4DataQueueException

peek

```
virtual void peek(const char* key,
                  IString& readString,
                  unsigned short keyLength =0,
                  long waitTime =0,
                  IString* senderInformation =NULL,
                  SearchType search =Search_Null);
virtual void peek(const char* key,
                  char* readBuffer,
                  unsigned long readBufferLength,
                  unsigned short keyLength =0,
                  long waitTime =0,
                  char* senderInformation =NULL,
                  SearchType search =Search_Null);
virtual void peek(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL);
virtual void peek(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL);
```

Reads an entry from an AS/400 data queue without removing the entry from the queue.

When a version that does not require a key is used, the key specified on the last `setKey()` is used to select the entry. If the key value was never set, a blank key is used.

If you use a version that does not have `search` or if you specify `Search_Null`, one of the following is used:

- The value from the last `setSearchType` is used.
- If `setSearchType` was never done, `Search_EQ` is used.

The parameters are the following:

key	(in) Key value to use for comparison when selecting an entry.
readString	(out) String to receive the entry.

readBuffer	(out) User-allocated storage to receive the entry.
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
keyLength	(in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created.
waitTime	(in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue.
senderInformation	(out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.
search	(in) The comparison to use when searching for a specific key. This is an optional parameter. If `Search_Null` is passed, the value from the last `setSearchType()` is used. If `setSearchType()` has not been done, *Search_EQ* is used.

Exceptions

- IC4DataQueueException

read

```
virtual void read(const char* key,
                  IString& readString,
                  unsigned short keyLength =0,
                  long waitTime =0,
                  IString* senderInformation =NULL,
                  SearchType search =Search_Null);
virtual void read(const char* key,
                  char* readBuffer,
                  unsigned long readBufferLength,
                  unsigned short keyLength =0,
                  long waitTime =0,
                  char* senderInformation =NULL,
                  SearchType search =Search_Null);
virtual void read(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL);
virtual void read(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL);
```

Reads an entry from the data queue. When a version that does not require a key is used, the key specified on the last `setKey()` is used to select the entry. If the key value was never set, a blank key is used.

If you use a version that does not have `search` or if you specify `Search_Null`, one of the following is used:

- The value from the last `setSearchType` is used.
- If `setSearchType` was never done, `Search_EQ` is used.

The parameters are the following:

key	(in) Key value to use for comparison when selecting an entry.
readString	(out) String to receive the entry.
readBuffer	(out) User-allocated storage to receive the entry.
readBufferLength	(in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur.
keyLength	(in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created.
waitTime	(in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue.
senderInformation	(out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned.
search	(in) The comparison to use when searching for a specific key. This is an optional parameter. If `Search_Null` is passed, the value used from the last `setSearchType()` is used. If `setSearchType()` has not been done, *Search_EQ* is used.

Exceptions

- IC4DataQueueException

searchType

```
virtual SearchType searchType() const;
```

Returns the value that describes the type of comparison used when searching for a specific key. For information on `SearchType`, see “SearchType” on page 101.

setConvertKey

```
void setConvertKey(IBoolean convertKey);
```

Set whether the key is converted between local system data format and EBCDIC when reading or writing an entry.

setKey

```
virtual void setKey(const char* key,
                    unsigned short keyLength =0);
```

Sets the key value used for comparisons when selecting an entry.

The parameters are the following:

key	(in) Key value to use for comparison
keyLength	(in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created.

Exceptions

- IC4DataQueueException

setSearchType

```
virtual void setSearchType(SearchType searchType);
```

Sets the value that describes the type of comparison used when searching for a specific key.

The parameters are the following:

searchType	(in) A SearchType enumeration value. For information on `SearchType`, see "SearchType" on page 101.

write

```
virtual void write(const char* key,
                   const char* writeString,
                   unsigned short keyLength =0,
                   unsigned long writeStringLength =0,
                   IBoolean commit =True);
virtual void write(const char* writeString,
                   unsigned long writeStringLength =0,
                   IBoolean commit =True);
```

Writes an entry to the data queue. When a version that does not require a key is used, the key specified on the last `setKey()` is used to write the entry. If the key value was never set, a blank key is used.

The parameters are the following:

key	(in) Value to use for key when putting an entry on the data queue.
writeString	(in) Entry to be put on data queue
keyLength	(in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created.
writeStringLength	(in) Size of the *writestring*. If this value is 0, *writestring* must be null-terminated.

commit	(in) True if the data is to be committed on the write before this function returns.

Exceptions

- IC4DataQueueException

Inherited Public Members

Member	Class	Page
clear	IC4BaseDataQueue	80
close	IC4BaseDataQueue	80
convert	IC4BaseDataQueue	80
dataLength	IC4BaseDataQueue	80
destroy	IC4BaseDataQueue	81
force	IC4BaseDataQueue	81
isOpen	IC4BaseDataQueue	81
maxRecordLength	IC4BaseDataQueue	81
name	IC4BaseDataQueue	81
operator==	IC4BaseDataQueue	82
operator!=	IC4BaseDataQueue	82
operator<	IC4BaseDataQueue	82
operator>	IC4BaseDataQueue	82
order	IC4BaseDataQueue	83
senderID	IC4BaseDataQueue	84
setConvert	IC4BaseDataQueue	84
systemName	IC4BaseDataQueue	84
textDescription	IC4BaseDataQueue	85

Enumerations

The following describes the SearchType enumeration.

SearchType

```
enum SearchType { Search_Null=0,
                  Search_EQ,
                  Search_NE,
                  Search_GT,
                  Search_GE,
                  Search_LT,
                  Search_LE };
```

Use these values for the data queue attribute that determines how entries are selected when comparing keys. *Search_Null* indicates that the last value set by `setSearchType()` should be used.

IC4KeyedDataQueue

Chapter 5. Data Translation Class

IC4Translate

The `IC4Translate` class provides a class interface to convert data from one format to another. For example, member functions are provided to convert ASCII data to EBCDIC data.

The data translation class uses AS/400 Client Access support for some data conversions. When a data translation member function receives a return code from AS/400 Client Access indicating that an error occurred during the conversion, an exception is thrown and the return code value appears in the exception text. For an explanation of the return codes refer to the appropriate documentation:

- For the AS/400 Client Access for Windows 95/NT client, see the `cwbdt.h` and `cwbnlcnv.h` header files.

Derivation

`IC4Translate` does not derive from another class.

Header File

`ic4xlt.hpp`

Constructors

All member functions for IC4Translate are static. Constructing an instance is not allowed.

Public Members

a2bin2

```
static IString a2bin2(const char* source);
static void a2bin2(IString& target,
                   const char* source);
static void a2bin2(char* target,
                   const char* source);
```

Converts an ASCII numeric string to a 2-byte integer in AS/400 format.

The parameters are the following:

source	(in) ASCII digits optionally preceded by a sign. Whitespace may precede the sign as in the C standard library function `atoi()`.
target	(out) Target buffer to receive converted value. An exception is thrown if the converted value overflows the target. If the source cannot be interpreted as a number, 0x0000 is returned.

Exceptions

- IC4TranslateException

a2bin4

```
static IString a2bin4(const char* source);
static void a2bin4(IString& target,
                   const char* source);
static void a2bin4(char* target,
                   const char* source);
```

Converts an ASCII numeric string to a 4-byte integer in AS/400 format.

The parameters are the following:

source	(in) ASCII digits optionally preceded by a sign. Whitespace may precede the sign as in the C standard library function `atol()`.
target	(out) Target buffer to receive converted value. An exception is thrown if the converted value overflows the target. If the source cannot be interpreted as a number, 0x00000000 is returned.

Exceptions

- IC4TranslateException

a2e

```
static IString a2e(const char* source,
                   unsigned long targetLength =0,
                   unsigned long sourceLength =0);
static void a2e(IString& target,
                const char* source,
                unsigned long targetLength =0,
                unsigned long sourceLength =0);
static void a2e(char* target,
                const char* source,
                unsigned long targetLength,
                unsigned long sourceLength =0);
static void a2e(char* target,
                const char* source,
                const char* systemName,
                unsigned long targetLength,
                unsigned long sourceLength);
```

Converts an ASCII character string to an EBCDIC string. Translation stops if a null character is encountered in the source. The target is blank padded.

A connection to an AS/400 system is needed to determine the target code page so a connection must be active before calling one of these member functions. A connection to the Client Access default system is required when using a member function that does not have a system name parameter. A connection to the specified system is required when using the member function that requires a system name parameter. The PC code page is used as the source code page.

The parameters are the following:

source	(in) ASCII string to convert.
sourceLength	(in) Length of source string. If the length is 0, the source must be null-terminated.
target	(out) Target buffer to receive converted value.
targetLength	(in) Length of the target area. The length must be greater than 0 when the target is a char*. When the target is an IString, the length defaults to 0. If the length is 0 *sourceLength* is used.
systemName	(in) The name of the AS/400 system for the target code page.

Exceptions

- IC4TranslateException

a2hex

```
static IString a2hex(const char* source,
                     unsigned long length =0);
static void a2hex(IString& target,
                  const char* source,
                  unsigned long length =0);
static void a2hex(char* target,
                  const char* source,
                  unsigned long length);
```

Converts an ASCII representation of a hexadecimal number to hexadecimal data.

The parameters are the following:

source	(in) ASCII string to convert. The value must: • Contain only characters from 0-9, A-F, a-f • Meet the length restrictions of the API being used
target	(out) Target buffer to receive converted value.
length	(in) Number of bytes of converted hexadecimal data to return. The length must be greater than 0 when the target is a char*. When the target is an IString, the length defaults to 0. If the length is 0, the source must be null-terminated and the number of bytes returned is one-half the length of the source.

Exceptions

- IC4TranslateException

a2pak

```
static IString a2pak(const char* source,
                     unsigned long length,
                     unsigned long decimalPosition =0);
static void a2pak(IString& target,
                  const char* source,
                  unsigned long length,
                  unsigned long decimalPosition =0);
static void a2pak(char* target,
                  const char* source,
                  unsigned long length,
                  unsigned long decimalPosition =0);
```

Converts an ASCII numeric string to EBCDIC packed decimal.

The parameters are the following:

source	(in) ASCII numeric string to convert. The value must: • Contains only characters 0-9, '+', '-', ' ', or '.' • Meet the length restrictions of the API being used
target	(out) Target buffer to receive converted value.
length	(in) Length in bytes of the packed decimal output to return. The API being used may document a restriction on the length.
decimalPosition	(in) Number of decimal digits in source string.

Exceptions

- IC4TranslateException

a2zon

```
static IString a2zon(const char* source,
                     unsigned long length,
                     unsigned long decimalPosition =0);
static void a2zon(IString& target,
                  const char* source,
                  unsigned long length,
                  unsigned long decimalPosition =0);
static void a2zon(char* target,
                  const char* source,
                  unsigned long length,
                  unsigned long decimalPosition =0);
```

Converts an ASCII numeric string to an EBCDIC zoned decimal.

The parameters are the following:

source	(in) ASCII numeric to convert. The value must: • Contain only characters 0-9, '+', '-', ' ', or '.' • Meet the length restrictions of the API being used
target	(out) Target buffer to receive converted value.
length	(in) Length in bytes of the zoned decimal number to return. The API being used may document a restriction on the length.

decimalPosition	(in) Number of decimal digits in source. Possible values are zero to *length*.

Exceptions

- IC4TranslateException

bin22a

```
static IString bin22a(const char* source);
static void bin22a(IString& target,
                   const char* source);
static void bin22a(char* target,
                   const char* source);
```

Converts a 2-byte EBCDIC binary number to a null-terminated ASCII numeric string.

The parameters are the following:

source	(in) 2-byte EBCDIC binary number to convert.
target	(out) Target buffer to receive converted value. Must be at least 7 bytes long to hold the largest possible value.

Exceptions

- IC4TranslateException

bin42a

```
static IString bin42a(const char* source);
static void bin42a(IString& target,
                   const char* source);
static void bin42a(char* target,
                   const char* source);
```

Converts a 4-byte EBCDIC binary number to a null-terminated ASCII numeric string.

The parameters are the following:

source	(in) A 4-byte EBCDIC binary number to convert.
target	(out) Target buffer to receive converted value. Must be at least 12 bytes long to hold the largest possible value.

Exceptions

- IC4TranslateException

dos2pak

```
static IString dos2pak(const char* source,
                       unsigned long length);
static void dos2pak(IString& target,
                    const char* source,
                    unsigned long length);
static void dos2pak(char* target,
                    const char* source,
                    unsigned long length);
```

Converts an ASCII packed decimal to an EBCDIC packed decimal.

The parameters are the following:

source	(in) ASCII packed decimal to convert.
target	(out) Target buffer to receive converted value.
length	(in) Number of bytes of the ASCII packed decimal source data to convert. The API being used may document a restriction on the length.

Exceptions

- IC4TranslateException

dos2zon

```
static IString dos2zon(const char* source,
                       unsigned long length);
static void dos2zon(IString& target,
                    const char* source,
                    unsigned long length);
static void dos2zon(char* target,
                    const char* source,
                    unsigned long length);
```

Converts an ASCII zoned decimal to an EBCDIC zoned decimal.

The parameters are the following:

source	(in) ASCII zoned decimal to convert.
target	(out) Target buffer to receive converted value.
length	(in) Number of bytes of the ASCII zoned decimal source to convert. The API being used may document a restriction on the length.

Exceptions

- IC4TranslateException

e2a

```
static IString e2a(const char* source,
                   unsigned long targetLength =0,
                   unsigned long sourceLength =0);
static void e2a(IString& target,
                const char* source,
                unsigned long targetLength =0,
                unsigned long sourceLength =0);
static void e2a(char* target,
                const char* source,
                unsigned long targetLength,
                unsigned long sourceLength =0);
static void e2a(char* target,
                const char* source,
                const char* systemName,
                unsigned long targetLength,
                unsigned long sourceLength);
```

Converts an EBCDIC string to an ASCII string. Translation stops if a null character is encountered in the source. The target is blank padded.

A connection to an AS/400 system is needed to determine the EBCDIC source code page so a connection must be active before calling one of these member functions. A connection to the Client Access default system is required when using a member function that does not have a system name parameter. A connection to the specified system is required when using the member function that requires a system name parameter. The PC code page is used as the target code page.

The parameters are the following:

source	(in) EBCDIC string to convert.
sourceLength	(in) Length of the source. If the length is 0, the source must be null-terminated.
target	(out) Target buffer to receive converted value.
targetLength	(in) Length of the target buffer. The length must be greater than 0 when the target is a char*. When the target is an IString, the length defaults to 0. If the length is 0, *sourceLength* is used.
systemName	(in) The name of the AS/400 system for the source code page.

Exceptions

- IC4TranslateException

e2e

```
static IString e2e(const char* source,
                   unsigned long length =0);
static void e2e(IString& target,
                const char* source,
                unsigned long length =0);
static void e2e(char* target,
                const char* source,
                unsigned long length);
```

Copies an EBCDIC string to another EBCDIC string. If data outside the range of x'40' to x'FF' is found in the source, only the data before the first data outside the range is copied.

The parameters are the following:

source	(in) EBCDIC string to copy.
target	(out) Target buffer to receive copied value.
length	(in) Number of bytes to copy. The API being used may document a restriction on the length. The length must be greater than 0 when the target is a char*. When the target is an IString, the length defaults to 0. If the length is 0, the source must be null-terminated.

Exceptions

- IC4TranslateException

hex2a

```
static IString hex2a(const char* source,
                      unsigned long length);
static void hex2a(IString& target,
                   const char* source,
                   unsigned long length);
static void hex2a(char* target,
                   const char* source,
                   unsigned long length);
```

Converts hexadecimal data to the ASCII representation of a hexadecimal.

The parameters are the following:

source	(in) Hexadecimal value to convert.
target	(out) Target buffer to receive converted value. Must be at least twice as long as source.
length	(in) Length of source. The API being used may document a restriction on the length.

Exceptions

- IC4TranslateException

pak2a

```
static IString pak2a(const char* source,
                      unsigned long length,
                      unsigned long decimalPosition =0);
static void pak2a(IString& target,
                   const char* source,
                   unsigned long length,
                   unsigned long decimalPosition =0);
static void pak2a(char* target,
                   const char* source,
                   unsigned long length,
                   unsigned long decimalPosition =0);
```

Converts an EBCDIC packed decimal to an ASCII numeric string.

The parameters are the following:

source	(in) EBCDIC packed decimal to convert.
target	(out) Target buffer to receive converted value.
length	(in) Length of source in bytes.
decimalPosition	(in) Number of decimal digits in target.

Exceptions

- IC4TranslateException

pak2dos

```
static IString pak2dos(const char* source,
                        unsigned long length);
static void pak2dos(IString& target,
                     const char* source,
                     unsigned long length);
static void pak2dos(char* target,
                     const char* source,
                     unsigned long length);
```

Converts an EBCDIC packed decimal to ASCII packed decimal.

The parameters are the following:

source	(in) EBCDIC packed decimal to be converted.
target	(out) Target buffer to receive converted value.
length	(in) Length of source.

Exceptions

- IC4TranslateException

pak2pak

```
static IString pak2pak(const char* source,
                        unsigned long length);
static void pak2pak(IString& target,
                     const char* source,
                     unsigned long length);
static void pak2pak(char* target,
                     const char* source,
                     unsigned long length);
```

Copies a packed decimal to a packed decimal. If invalid packed decimal data is found, only the data before the invalid data is copied

The parameters are the following:

source	(in) Packed decimal to copy.
target	(out) Target buffer to receive copied value.
length	(in) Number of bytes to copy.

Exceptions

- IC4TranslateException

zon2a

```
static IString zon2a(const char* source,
                       unsigned long length,
                       unsigned long decimalPosition =0);
static void zon2a(IString& target,
                    const char* source,
                    unsigned long length,
                    unsigned long decimalPosition =0);
static void zon2a(char* target,
                    const char* source,
                    unsigned long length,
                    unsigned long decimalPosition =0);
```

Converts an EBCDIC zoned decimal to an ASCII numeric string.

The parameters are the following:

source	(in) EBCDIC zoned decimal to convert.
target	(out) Target buffer to receive converted value. Must be large enough to hold the sign and decimal point.
length	(in) Length of source in bytes.
decimalPosition	(in) Number of decimal digits in target.

Exceptions

- IC4TranslateException

zon2dos

```
static IString zon2dos(const char* source,
                         unsigned long length);
static void zon2dos(IString& target,
                      const char* source,
                      unsigned long length);
static void zon2dos(char* target,
                      const char* source,
                      unsigned long length);
```

Converts an EBCDIC zoned decimal to an ASCII zoned decimal.

The parameters are the following:

source	(in) EBCDIC zoned decimal to convert.
target	(out) Target buffer to receive converted value.
length	(in) Number of bytes to convert.

Exceptions

- IC4TranslateException

zon2zon

```
static IString zon2zon(const char* source,
                        unsigned long length);
static void zon2zon(IString& target,
                    const char* source,
                    unsigned long length);
static void zon2zon(char* target,
                    const char* source,
                    unsigned long length);
```

Copies a zoned decimal to another zoned decimal.

The parameters are the following:

source	(in) Zoned decimal to copy.
target	(out) Target buffer to receive copied value.
length	(in) Number of bytes to copy.

Exceptions

- IC4TranslateException

IC4Translate

Chapter 6. Data Types Classes

The data types class group can be used to work with AS/400 data on a personal computer. Because the AS/400 and the PC use different formats for data, AS/400 data usually must be converted to a PC representation before it can be used by a C++ program on the PC. This class group provides several classes that represent AS/400 data types as common C++ data types. For example, an AS/400 2-byte binary number can be used as a `short int`. Conversion functions allow converting data between the AS/400 and PC formats, including translating character data between EBCDIC and ASCII. Aggregate classes can be used when more complex structures must be built from the basic data types.

IC4Array

The `IC4Array` class allows data items of the same type to be worked with as an array. `IC4Array` uses the default constructor for the element type when constructing an array. The default constructor for some data types leaves the new object in an incomplete state. For example, `IC4Char` takes a *size* argument, and some data types, such as `IC4Struct`, cannot be completed during construction. In cases where the object is left incomplete by the default constructor, `initializeElements()` must be called to complete the initialization.

Derivation

```
IC4DataType
 IC4CPDDataType
  IC4ArrayBase
   IC4Array
```

Header File

`ic4array.hpp`

Constructors

```
IC4Array();
IC4Array(unsigned long elementCount);
IC4Array(unsigned long elementCount,
         void* dataPointer,
         unsigned long dataSize,
         void* externalDataPointer =NULL,
         unsigned long externalDataSize =0);
IC4Array(const IC4Array<Type>& other);
```

The parameters are the following:

elementCount	(in) The number of elements in the array. Must be greater than zero. If element count is not specified, then the number of elements must be set using `setElementCount()` before any other `IC4Array` member functions are called.

dataPointer	(in) A pointer to where the local format data for elements of the array is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
dataSize	(in) The size of the local format data. This value is the maximum size of the local format data for the `IC4Array` object.
externalDataPointer	(in) A pointer to where the external format data for elements of the array is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.
externalDataSize	(in) The size of the external format data. This value is the maximum size of the external format data for the `IC4Array` object.
other	(in) An `IC4Array` object that is to be copied. The array is copied by value.

Exceptions

- IC4DataTypeException

Public Members

clone

`virtual IC4DataType* **clone**() const;`

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

initializeElements

`void **initializeElements**(const Type& *source*);`

Initializes the elements of the array to the value of the *source* data item. Both the local and external format data values are copied. This member function must be called if a data item constructed using the default constructor is not complete. The default constructors of the following classes leave the data item in an incomplete state:

`IC4Byte`
`IC4Char`
`IC4Array` of `IC4Char` or `IC4Byte`

The parameters are the following:

source	(in) A data item to use for the initialization of each element.

Exceptions

- IC4DataTypeException

operator=

```
IC4Array<Type>& operator=(const IC4Array<Type>& other);
```

Assignment operator: Both `IC4Array` objects must have the same number of elements and the elements must be the same size.

The parameters are the following:

other	(in) An `IC4Array` object that is to be copied. The array is copied by value.

Exceptions

- IC4DataTypeException

operator []

```
const Type& operator[](unsigned long elementIndex) const;
Type& operator[](unsigned long elementIndex);
```

Returns a reference to the array element at elementIndex.

The parameters are the following:

elementIndex	(in) The 0-based index of an element in the array.

Exceptions

- IC4DataTypeException

setElementCount

```
void setElementCount(unsigned long elementCount);
```

Sets the number of elements in the array when the default constructor was used to construct the array. An exception is thrown if this member function is called more than once.

The parameters are the following:

elementCount	(in) The number of elements in the array.

Exceptions

- IC4DataTypeException

Inherited Public Members

Member	Class	Page
codePage	IC4CPDDataType	141
codePageOverrideExpected	IC4CPDDataType	141
dataPointer	IC4ArrayBase	118
dataSize	IC4ArrayBase	119

IC4ArrayBase

Member	Class	Page
elementCount	IC4ArrayBase	119
externalCodePage	IC4CPDDataType	142
externalDataPointer	IC4ArrayBase	119
externalDataSize	IC4ArrayBase	119
overlay	IC4DataType	145
resetCodePages	IC4CPDDataType	142
setCodePage	IC4CPDDataType	142
setCodePageOverrideExpected	IC4CPDDataType	142
setCodePages	IC4CPDDataType	142
setDataPointer	IC4ArrayBase	119
setExternalCodePage	IC4CPDDataType	143
setExternalDataPointer	IC4ArrayBase	120
translateFromExternal	IC4ArrayBase	120
translateToExternal	IC4ArrayBase	120

IC4ArrayBase

The `IC4ArrayBase` class is the base class for the array template class `IC4Array`. These two classes provide the capability to construct and use arrays that contain objects that are derived from `IC4DataType`. `IC4ArrayBase` implements array member functions that are not type specific.

An `IC4Array` object may have elements that are dependent on character set code pages for proper conversion between AS/400 and PC formats, and these elements may expect to have the code page values explicitly specified when conversion is done. See the base class, "IC4CPDDataType" on page 141, for additional information.

Derivation

```
IC4DataType
 IC4CPDDataType
  IC4ArrayBase
```

Header File

```
ic4array.hpp
```

Constructors

`IC4ArrayBase` is an abstract base class. Constructing an instance is not allowed.

Public Members

dataPointer

```
virtual void* dataPointer() const;
```

Returns a pointer to the local format data. An exception is thrown if `dataSize()` is zero and storage was not explicitly assigned.

Exceptions

- IC4DataTypeException

dataSize

```
virtual unsigned long dataSize() const;
```

Returns the size in bytes of the local format data.

elementCount

```
unsigned long elementCount() const;
```

Returns the number of elements in the array.

externalDataPointer

```
virtual void* externalDataPointer() const;
```

Returns a pointer to the external format data. An exception is thrown if `externalDataSize()` is zero and storage was not explicitly assigned.

Exceptions

- IC4DataTypeException

externalDataSize

```
virtual unsigned long externalDataSize() const;
```

Returns the size in bytes of the external format data.

setDataPointer

```
void setDataPointer(void* dataPointer,
                    unsigned long size =0);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.
size	(in) The size in bytes of the user-managed storage being assigned. Must be greater than zero and large enough to contain the current size of the local format data. This parameter is ignored if *dataPointer* is NULL.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(void* externalDataPointer,
                            unsigned long size =0);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.
size	(in) The size in bytes of the user-managed storage being assigned. Must be greater than zero and large enough to contain the current size of the external format data. This parameter is ignored if *externalDataPointer* is NULL.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
virtual void translateFromExternal(unsigned long codePage,
                                   unsigned long externalCodePage);
```

Converts the data from the external format to the local format. If code page values are specified, those values are used for elements that return True for `codePageOverrideExpected()`. Otherwise, the code pages of the elements are used.

The parameters are the following:

codePage	(in) The code page to use for the local format data.
externalCodePage	(in) The code page to use for the external format data.

Exceptions

- IC4DataTypeException

translateToExternal

```
virtual void translateToExternal();
virtual void translateToExternal(unsigned long codePage,
                                 unsigned long externalCodePage);
```

Converts the data to the external format from the local format. If code page values are specified, those values are used for elements that return True for `codePageOverrideExpected()`. Otherwise, the code pages of the elements are used.

The parameters are the following:

codePage	(in) The code page to use for the local format data.
externalCodePage	(in) The code page to use for the external format data.

Exceptions

- IC4DataTypeException

Inherited Public Members

Member	Class	Page
codePage	IC4CPDDataType	141
codePageOverrideExpected	IC4CPDDataType	141
externalCodePage	IC4CPDDataType	142
overlay	IC4DataType	145
resetCodePages	IC4CPDDataType	142
setCodePage	IC4CPDDataType	142
setCodePageOverrideExpected	IC4CPDDataType	142
setCodePages	IC4CPDDataType	142
setExternalCodePage	IC4CPDDataType	143

IC4Bin2

The `IC4Bin2` class represents an AS/400 2-byte binary number. For an `IC4Bin2` object, `ExternalFormat` is a typedef of char[2]. The local format is `short int`.

Derivation

```
IC4DataType
 IC4Bin2
```

Header File

```
ic4bin.hpp
```

Constructors

```
IC4Bin2();
```

The default constructor initializes the local format data to zero.

```
IC4Bin2(short value,
        ExternalFormat externalDataPointer =NULL);
IC4Bin2(short* dataPointer,
        ExternalFormat externalDataPointer =NULL);
IC4Bin2(const IC4Bin2& other);
```

The parameters are the following:

value	(in) The initial value for the local format data.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.
dataPointer	(in) A pointer to where the local format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
other	(in) An `IC4Bin2` object. The data item is copied by value.

Public Members

clone

`virtual IC4DataType* **clone**() const;`

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

dataPointer

`virtual void* **dataPointer**() const;`

Returns a pointer to the local format data.

Exceptions

- IC4DataTypeException

dataSize

`virtual unsigned long **dataSize**() const;`

Returns the size in bytes of the local format data.

externalDataPointer

`virtual void* **externalDataPointer**() const;`

Returns a pointer to the external format data.

Exceptions

- IC4DataTypeException

externalDataSize

`virtual unsigned long **externalDataSize**() const;`

Returns the size in bytes of the external format data.

operator short

```
operator short&();
operator short() const;
```

Conversion operators: Return a reference to the local format data or a copy of the local format data.

operator=

```
IC4Bin2& operator=(short value);
IC4Bin2& operator=(const IC4Bin2& other);
```

Assignment operators.

The parameters are the following:

value	(in) The value to assign to the local format data.
other	(in) An `IC4Bin2` object from which to retrieve the local and external format data values. The data item is copied by value.

setDataPointer

```
void setDataPointer(short* dataPointer);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(ExternalFormat externalDataPointer);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than

NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
```

Converts the data from the external format to the local format.

Exceptions

- IC4DataTypeException

translateToExternal

```
virtual void translateToExternal();
```

Converts the data to the external format from the local format.

Exceptions

- IC4DataTypeException

Inherited Public Members

Member	Class	Page
codePageOverrideExpected	IC4DataType	144
overlay	IC4DataType	145

IC4Bin4

The `IC4Bin4` class represents an AS/400 4-byte binary number. For an `IC4Bin4` object, `ExternalFormat` is a typedef of char[4]. The local format is `long int`.

Derivation

```
IC4DataType
  IC4Bin4
```

Header File

```
ic4bin.hpp
```

Constructors

```
IC4Bin4();
```

The default constructor initializes the local format data to zero.

```
IC4Bin4(long value,
        ExternalFormat externalDataPointer =NULL);
IC4Bin4(long* dataPointer,
        ExternalFormat externalDataPointer =NULL);
IC4Bin4(const IC4Bin4& other);
```

The parameters are the following:

value	(in) The initial value for the local format data.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.
dataPointer	(in) A pointer to where the local format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
other	(in) An `IC4Bin4` object. The data item is copied by value.

Public Members

clone

```
virtual IC4DataType* clone() const;
```

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

dataPointer

```
virtual void* dataPointer() const;
```

Returns a pointer to the local format data.

Exceptions

- IC4DataTypeException

dataSize

```
virtual unsigned long dataSize() const;
```

Returns the size in bytes of the local format data.

externalDataPointer

```
virtual void* externalDataPointer() const;
```

Returns a pointer to the external format data.

Exceptions

- IC4DataTypeException

externalDataSize

```
virtual unsigned long externalDataSize() const;
```

Returns the size in bytes of the external format data.

operator long

```
operator long&();
operator long() const;
```

Conversion operators: Return a reference to the local format data or a copy of the local format data.

operator=

```
IC4Bin4& operator=(long value);
IC4Bin4& operator=(const IC4Bin4& other);
```

Assignment operators.

The parameters are the following:

value	(in) The value to assign to the local format data.
other	(in) An `IC4Bin4` object from which to copy the local and external format data values. The data item is copied by value.

setDataPointer

```
void setDataPointer(long* dataPointer);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(ExternalFormat externalDataPointer);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
```

Converts the data from the external format to the local format.

Exceptions

- IC4DataTypeException

translateToExternal

```
virtual void translateToExternal();
```

Converts the data to the external format from the local format.

Exceptions

- IC4DataTypeException

Inherited Public Members

Member	Class	Page
codePageOverrideExpected	IC4DataType	144
overlay	IC4DataType	145

IC4Byte

The `IC4Byte` class represents a byte array that does not require conversion between AS/400 format and PC format. Use the `IC4Char` class if conversion between system formats is required. Because this data type does not support conversion, most instances of `IC4Byte` do not have local format data.

Derivation

```
IC4DataType
  IC4Byte
```

Header File

`ic4byte.hpp`

Constructors

To construct an `IC4Byte` object without allocating storage, use this version of the constructor. Before the data item can be added as a member of an `IC4Struct` object or used to initialize an `IC4Array<IC4Byte>` object, the size must be set using `initialize()` or `setDataValue()`.

```
IC4Byte();
```

These versions construct an `IC4Byte` object that has only external format data. Use one of these versions when translation of the data is not required.

```
IC4Byte(const void* value,
        unsigned long externalDataSize);
IC4Byte(unsigned long externalDataSize,
        void* externalDataPointer =NULL);
```

The parameters are the following:

value	(in) The initial value for the external format data.
externalDataSize	(in) The number of bytes of external format data. Must be greater than zero.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.

These versions construct an `IC4Byte` object that has both local format data and external format data. Use one of these versions when this data item will be overlaid by another data item that has local format data.

```
IC4Byte(const void* value,
         unsigned long dataSize,
         unsigned long externalDataSize,
         void* externalDataPointer =NULL);
IC4Byte(unsigned long dataSize,
         unsigned long externalDataSize,
         void* dataPointer =NULL,
         void* externalDataPointer =NULL);
```

The parameters are the following:

value	(in) The initial value for the local format data.
dataSize	(in) The number of bytes of local format data. Must be greater than zero.
externalDataSize	(in) The number of bytes of external format data. Must be greater than zero.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.
dataPointer	(in) A pointer to where the local format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.

To construct an `IC4Byte` object by copying another one, use this version.

```
IC4Byte(const IC4Byte& other);
```

The parameters are the following:

other	(in) An `IC4Byte` object. The data item is copied by value.

Public Members

clone

```
virtual IC4DataType* clone() const;
```

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

dataPointer

```
virtual void* dataPointer() const;
```

Returns a pointer to the local format data.

Exceptions

- IC4DataTypeException

dataSize

```
virtual unsigned long dataSize() const;
```

Returns the size in bytes of the local format data.

externalDataPointer

```
virtual void* externalDataPointer() const;
```

Returns a pointer to the external format data.

Exceptions

- IC4DataTypeException

externalDataSize

```
virtual unsigned long externalDataSize() const;
```

Returns the size in bytes of the external format data.

initialize

This version completes the initialization of data items that have only external format data. When initialization is complete, the data item is in the same state as if it was constructed by the corresponding version of the constructor. Any previously assigned storage is no longer assigned.

```
void initialize(const void* value,
                unsigned long externalDataSize);
void initialize(unsigned long externalDataSize,
                void* externalDataPointer =NULL);
```

The parameters are the following:

value	(in) A pointer to the value for the external format data.
externalDataSize	(in) The number of bytes of external format data. Must be greater than zero.
externalDataPointer	(in) A pointer to where the external format data is to be stored. If not specified, the external format data is in object-managed storage.

This version completes the initialization of data items that have both local format data and external format data. When initialization is complete, the data item is in the same state as if it was constructed by the corresponding version of the constructor. Any previously assigned storage is no longer assigned.

```
void initialize(const void* value,
                unsigned long dataSize,
                unsigned long externalDataSize,
                void * externalDataPointer =NULL);
void initialize(unsigned long dataSize,
                unsigned long externalDataSize,
                void * dataPointer =NULL,
                void* externalDataPointer =NULL);
```

The parameters are the following:

value	(in) A pointer to the value for the external format data.
dataSize	(in) Number of bytes of local format data. Must be greater than zero.
externalDataSize	(in) The number of bytes of external format data. Must be greater than zero.
externalDataPointer	(in) A pointer to where the external format data is to be stored. If not specified, the external format data is in object-managed storage.
dataPointer	(in) A pointer to where the local format data is to be stored. If not specified, the local format data is in object-managed storage.

Exceptions

- IC4DataTypeException

operator=

```
IC4Byte& operator=(const IC4Byte& other);
```

Assignment operator: Copies the local format and external format data from another `IC4Byte` object. After the assignment is complete, the local and external format data sizes of this data item will match those of the data item that was copied. An exception is thrown if data will be lost.

If this data item has a fixed size, the local format and external format data sizes must match those of the copied item.

The parameters are the following:

other	(in) An `IC4Byte` object from which to retrieve the value and size for the local and external format data. The data item is copied by value.

Exceptions

- IC4DataTypeException

operator void*

```
operator void*();
operator const void*() const;
```

Conversion operators: Return a pointer to the external format data. An exception is thrown if the data item:

- Is contained in an `IC4Struct` object and structure storage has not been assigned

- Is not contained in an `IC4Struct` object, has an external data size of zero, and user-managed storage has not been assigned to hold the external format data

Exceptions

- IC4DataTypeException

setDataPointer

```
void setDataPointer(void* dataPointer,
                    unsigned long dataSize);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.
dataSize	(in) The size of the user-managed storage where the local format data is to be stored. Must be greater than zero. This parameter is ignored if *dataPointer* is NULL.

Exceptions

- IC4DataTypeException

setDataValue

```
void setDataValue(const void* value,
                  unsigned long size);
```

Sets the value of the external format data. An exception is thrown If the data item is contained (for example, it is a member of an `IC4Struct` object) and the value specified for *size* would change the external data size of this item.

The parameters are the following:

value	(in) A pointer to the value for the data item. Passing NULL causes the number of bytes specified by *size* to be set to null characters.
size	(in) The number of bytes of data to set. Must be greater than zero. The external format data size is set to this value.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(void* externalDataPointer,
                            unsigned long externalDataSize);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.
externalDataSize	(in) The size of the user-managed storage where the external format data is to be stored. Must be large enough to hold the data item. This parameter is ignored if *externalDataPointer* is NULL.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
```

This member function does not do any conversion for instances of this class.

translateToExternal

```
virtual void translateToExternal();
```

This member function does not do any conversion for instances of this class.

Inherited Public Members

Member	Class	Page
codePageOverrideExpected	IC4DataType	144
overlay	IC4DataType	145

IC4Char

The IC4Char class represents an AS/400 text string that requires conversion using character set code pages. Use the IC4Byte class if conversion is not required.

An IC4Char object may expect to have code page values explicitly specified when conversion between AS/400 and PC formats is done. See the base class, “IC4CPDDataType” on page 141, for more information.

IC4Char

Derivation

```
IC4DataType
 IC4CPDDataType
  IC4Char
```

Header File

`ic4char.hpp`

Constructors

Use this constructor to create an `IC4Char` object without setting a size and allocating storage. The data sizes will be set to zero.

```
IC4Char();
```

Use this constructor to create an `IC4Char` object with an initial local format data value in object-managed storage. Optionally, the size of the external format data can be set and user-managed storage can be assigned to hold the external format data.

```
IC4Char(const char* value,
        unsigned long externalDataSize =0,
        char* externalDataPointer =NULL);
```

The parameters are the following:

value	(in) A null-terminated initial value for the local format data. The size of the data item is set to the size of the initial value and the null-terminator character is stored with the data but is discarded if this data item is later contained by an `IC4Struct` object.
externalDataSize	(in) The number of bytes of external format data. Must be greater than zero if *externalDataPointer* is not NULL. If this value is greater than zero, then the external data size is fixed at this size.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.

Use this constructor to create an `IC4Char` object with the specified size. Optionally, user-managed storage can be assigned to hold the local and external format data.

```
IC4Char(unsigned long dataSize,
        unsigned long externalDataSize =0,
        char* dataPointer =NULL,
        char* externalDataPointer =NULL);
```

The parameters are the following:

dataSize	(in) Number of bytes of local format data. Must be greater than zero if *dataPointer* is not NULL. If this value is zero then *externalDataSize* must be greater than zero.

externalDataSize	(in) The number of bytes of external format data. Must be greater than zero if *externalDataPointer* is not NULL. If this value is greater than zero, then the external data size is fixed at this size.
dataPointer	(in) A pointer to where the local format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.

Use this constructor to create an `IC4Char` object by copying another `IC4Char` object.

```
IC4Char(const IC4Char& other);
```

The parameters are the following:

other	(in) An `IC4Char` object. The data item is copied by value.

Exceptions

- IC4DataTypeException

Public Members

clone

```
virtual IC4DataType* clone() const;
```

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

dataPointer

```
virtual void* dataPointer() const;
```

Returns a pointer to the local format data. If storage has not been assigned and `dataSize()` returns zero, a single byte of storage is allocated and initialized to the null-terminator character.

Exceptions

- IC4DataTypeException

dataSize

```
virtual unsigned long dataSize() const;
```

Returns the size in bytes of the local format data.

externalDataPointer

```
virtual void* externalDataPointer() const;
```

Returns a pointer to the external format data.

If the data item has an external data size of zero and has not been assigned user-managed storage for external format data, an exception is thrown.

Exceptions

- IC4DataTypeException

externalDataSize

```
virtual unsigned long externalDataSize() const;
```

Returns the size in bytes of the external format data.

initialize

```
void initialize(char* value,
                unsigned long externalDataSize =0,
                char* externalDataPointer =NULL);
void initialize(unsigned long dataSize,
                unsigned long externalDataSize =0,
                char* dataPointer =NULL,
                char* externalDataPointer =NULL);
```

Completes the initialization for a data item that was constructed using the default constructor.

The parameters are the following:

value	(in) A null-terminated initial value for the local format data. The size of the data item is set to the size of the initial value, the null-terminator character is stored with the data in storage managed by this data item. The null-terminator character is discarded if this object is later contained by an `IC4Struct` object.
externalDataSize	(in) The number of bytes of external format data. Must be greater than zero if *externalDataPointer* is not NULL.
externalDataPointer	(in) A pointer to where the external format data is to be stored.
dataSize	(in) Number of bytes of local format data. Must be greater than zero if *dataPointer* is not NULL.
dataPointer	(in) A pointer to where the local format data is to be stored.

Exceptions

- IC4DataTypeException

operator=

```
IC4Char& operator=(const char* value);
IC4Char& operator=(const IC4Char& other);
```

Assignment operators: If the assignment will result in the loss of data, an exception is thrown. This can happen if this data item is contained in an IC4Struct or IC4Array data item or if user-managed storage was assigned and the `IC4Char` object being copied is too large for this data item.

If the size of this data item is fixed (for example, if it is contained in an IC4Struct object) and the size of the value being copied is less than the size of this data item, an exception is thrown.

If this data item is not contained, its size is adjusted to equal the length of *value*. A null character follows the copied value.

The parameters are the following:

value	(in) The value to assign to the local format data.
other	(in) An `IC4Char` object from which to assign the local and external format data values. The data item is copied by value.

operator char*

```
operator char*();
operator const char*() const;
```

Conversion operators: Return a pointer to the local format data. If storage has not been assigned and `dataSize()` returns zero, a single byte of storage is allocated and initialized to the null-terminator character.

Exceptions

- IC4DataTypeException

setDataPointer

```
void setDataPointer(char* dataPointer,
                    unsigned long dataSize);
```

Assigns user-managed storage to hold the local format data.

The parameters are the following:

dataPointer	(in) Pointer to storage for local format data. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.

dataSize	(in) The size in bytes of the user-managed storage where the local format data is stored. When *dataPointer* is NULL, this parameter is ignored.

Exceptions

- IC4DataTypeException

setDataValue

```
void setDataValue(const char* value,
                  unsigned long dataSize);
```

Sets the local format data value. Use this member function for values that may contain null characters. If this member function will cause the loss of data, an exception is thrown. This can happen if user-managed storage was assigned or this data item is contained in an `IC4Struct` or `IC4Array` object, and the value being copied is too large.

If the size of this data item is fixed (for example if it is contained in an `IC4Struct` object) and the size of the value being copied is less than the size of this data item, an exception is thrown. If this data item is not contained, its size is set to the size of *value*.

The parameters are the following:

value	(in) A value for the local format data.
dataSize	(in) Length in bytes of *value*.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
virtual void setExternalDataPointer(char* externalDataPointer,
                                    unsigned long externalDataSize);
```

Assigns user-managed storage to hold the external format data.

The parameters are the following:

externalDataPointer	(in) Pointer to storage for the external format data. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.
externalDataSize	(in) The size in bytes of the user-managed storage where the external format data is stored. Must be greater than zero. When *externalDataPointer* is NULL, this value is ignored.

Exceptions

- IC4DataTypeException

setExternalPad

```
void setExternalPad(char pad ='\0');
void setExternalPad(const char* pad,
                    unsigned long padLength =0);
```

Sets a pad value for fixed length external format data and enables padding. This member function affects only the `translateToExternal()` member function.

The parameters are the following:

char pad	(in) Character to be used to pad. If '\0' is specified, no padding will be done.
char pad*	(in) Value to be used to pad the external format data. If NULL is specified, an exception is thrown.
padLength	(in) Length of the pad string. If this value is zero, then the pad string must be null-terminated.

Exceptions

- IC4DataTypeException

setGraphic

```
void setGraphic(IBoolean graphic =True);
```

Sets the attribute that describes how double-byte characters appear in the external format data. Double byte data can be *bracketed* or *graphic*. *Bracketed* means that sequences of double-byte characters are enclosed by shift-out and shift-in characters. *Graphic* means that the data is made up only of double-byte characters. The external format data is assumed to be bracketed unless this member function is used to change it.

The parameters are the following:

graphic	(in) True indicates that the external data format contains only double byte characters. False indicates that double byte characters are enclosed by shift-out and shift-in characters.

translatedBytes

```
unsigned long translatedBytes() const;
```

Returns the number of bytes that were converted during the most recent call to `translateFromExternal()` or `translateToExternal()`. This value is useful when a data item is contained in an `IC4Struct` or `IC4Array` object, and the number of bytes resulting from the conversion is less than the size of the local format data. Pad characters are not included in the value returned.

translateFromExternal

```
virtual void translateFromExternal();
virtual void translateFromExternal(unsigned long codePage,
                                   unsigned long externalCodePage);
```

Converts the data from the external format to the local format. If code page values are specified, those values are used for the conversion. Otherwise, the code pages of the data item are used. If the conversion will result in the loss of data, an exception is thrown. This can happen if user-managed storage was assigned or this data item is contained in an `IC4Struct` or `IC4Array` and the converted data is too large for the local format data.

The parameters are the following:

codePage	(in) The code page to use for the local format data.
externalCodePage	(in) The code page to use for the external format data.

Exceptions

- IC4DataTypeException

translateToExternal

```
virtual void translateToExternal();
virtual void translateToExternal(unsigned long codePage,
                                 unsigned long externalCodePage);
```

Converts the data to the external format from the local format. If code page values are specified, those values are used for the conversion. Otherwise, the code pages of the data item are used. If the conversion will result in the loss of data, an exception is thrown. This can happen if user-managed storage was assigned was assigned or this data item is contained in an `IC4Struct` or `IC4Array` and the converted data is too large for the local format data.

The parameters are the following:

codePage	(in) The code page to use for local format data.
externalCodePage	(in) The code page to use for external format data.

Exceptions

- IC4DataTypeException

Inherited Public Members

Member	Class	Page
codePage	IC4CPDDataType	141
codePageOverrideExpected	IC4CPDDataType	141
externalCodePage	IC4CPDDataType	142
overlay	IC4DataType	145
resetCodePages	IC4CPDDataType	142
setCodePage	IC4CPDDataType	142

Member	Class	Page
setCodePageOverrideExpected	IC4CPDDataType	142
setCodePages	IC4CPDDataType	142
setExternalCodePage	IC4CPDDataType	143

IC4CPDDataType

The `IC4CPDDataType` class is the base class for derived classes that represent an AS/400 data type requiring character set code pages for conversion support. These data types are referred to as "code-page dependent" data types. This class provides functions for managing and using code page values for data items of a derived class. Classes that do not require conversion using code pages should derive from the `IC4DataType` class.

An instance of a code-page dependent data type can specify code page values on the constructor or by using one of the available member functions to set the code page values. A data item can request that its code page values be overridden when `translateToExternal` or `translateFromExternal` are called to convert data. If code page values are not specified, the conversion is done using the default English-to-English, ASCII-to/from-EBCDIC code pages.

`IC4CPDDataType` is the base class for `IC4Char`, `IC4Struct`, and `IC4ArrayBase`.

Derivation

```
IC4DataType
 IC4CPDDataType
```

Header File

`ic4dtcpd.hpp`

Constructors

`IC4CPDDataType` is an abstract base class. Constructing an instance is not allowed.

Public Members

codePage

```
unsigned long codePage() const;
```

Returns the code page value for the local format data.

codePageOverrideExpected

```
virtual IBoolean codePageOverrideExpected() const;
```

Returns True if the data item expects code page values to be specified when `translateToExternal` and `translateFromExternal` are called to convert data.

externalCodePage

```
unsigned long externalCodePage() const;
```

Returns the code page value for the external format data.

resetCodePages

```
void resetCodePages();
```

Resets the code page values to use the English-to-English ASCII to/from EBCDIC code pages.

setCodePage

```
void setCodePage(unsigned long codePage);
```

Sets the code page value for the local format data.

The parameters are the following:

codePage	(in) The code page to use for the local format data.

setCodePageOverrideExpected

```
virtual void setCodePageOverrideExpected(IBoolean expected =True);
```

Sets an indication of whether the data item expects code page values to be specified when `translateToExternal` and `translateFromExternal` are called to convert data.

The parameters are the following:

expected	(in) True indicates that the code pages should be overridden.

setCodePages

```
void setCodePages(unsigned long codePage,
                  unsigned long externalCodePage);
void setCodePages(const char* systemName);
```

Sets the code page values for both the local format data and the external format data.

The parameters are the following:

codePage	(in) The code page to use for the local format data.
externalCodePage	(in) The code page to use for the external format data.
systemName	(in) The name of the AS/400 system from which to retrieve the code page values. The default code page for the system is used for the external format data. The local code page is the PC default code page as determined by AS/400 Client Access. When this parameter is specified, a connection to the AS/400 system is established. If *systemName* is NULL, the default Client Access/400 system name is used.

Exceptions

- IC4DataTypeException

setExternalCodePage

```
void setExternalCodePage(unsigned long externalCodePage);
```

Sets the code page value for the external format data.

The parameters are the following:

externalCodePage	(in) The code page to use for the external format data.

translateFromExternal

```
virtual void translateFromExternal() =0;
virtual void translateFromExternal(unsigned long codePage,
                                   unsigned long externalCodePage) =0;
```

Converts the data from external format to local format. If code page values are not specified and were not explicitly set for the data item, then the default English-to-English EBCDIC to ASCII conversion is done.

The parameters are the following:

codePage	(in) The code page to use for the local format data.
externalCodePage	(in) The code page to use for the external format data.

Exceptions: See derived class member function.

translateToExternal

```
virtual void translateToExternal() =0;
virtual void translateToExternal(unsigned long codePage,
                                 unsigned long externalCodePage) =0;
```

Converts the data to external format from local format. If code page values are not specified and were not explicitly set for the data item, then the default English-to-English ASCII to EBCDIC conversion is done.

The parameters are the following:

codePage	(in) The code page to use for the local format data.
externalCodePage	(in) The code page to use for the external format data.

Exceptions: See derived class member function.

Inherited Public Members

Member	Class	Page
clone	IC4DataType	144
dataPointer	IC4DataType	144

IC4Datatype

Member	Class	Page
dataSize	IC4DataType	145
externalDataPointer	IC4DataType	145
externalDataSize	IC4DataType	145
overlay	IC4DataType	145

IC4DataType

The `IC4DataType` class is a base class for derived classes that represent specific data types. Data types derived from IC4DataType are used to translate data from one format to another.

Derivation

`IC4DataType` does not derive from another class.

Header File

`ic4dt.hpp`

Constructors

IC4DataType is an abstract base class. Constructing an instance is not allowed.

Public Members

clone

```
virtual IC4DataType* clone() const=0;
```

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

codePageOverrideExpected

```
virtual IBoolean codePageOverrideExpected() const;
```

Returns True if either of the following are true:

- Is dependent on character set code page values for correct conversion
- Expects code page values to be specified when `translateToExternal()` or `translateFromExternal()` are called to convert data

If not overridden by a derived class, this member function returns false. See the `IC4CPDDataType` class for more information about overriding code pages.

dataPointer

```
virtual void* dataPointer() const=0;
```

Returns a pointer to the local format data.

Exceptions: See derived class member function.

dataSize

```
virtual unsigned long dataSize() const=0;
```

Returns the size in bytes of the local format data.

Exceptions: See derived class member function.

externalDataPointer

```
virtual void* externalDataPointer() const=0;
```

Returns a pointer to the external format data.

Exceptions: See derived class member function.

externalDataSize

```
virtual unsigned long externalDataSize() const=0;
```

Returns the size in bytes of the external format data.

Exceptions: See derived class member function.

overlay

```
void overlay(const IC4DataType& other);
```

Maps this data item over the local and external format data of another data item. Upon return from this member function, the storage locations of this data item are pointing to the same locations that the other data item's storage pointers are pointing at. No bounds checking is performed.

The parameters are the following:

other	(in) The data item that is to be overlaid. The data item is copied by value.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal() =0;
```

Converts the data from the external format to the local format.

Exceptions: See derived class member function.

translateToExternal

```
virtual void translateToExternal() =0;
```

Converts the data to the external format from the local format.

Exceptions: See derived class member function.

IC4Float4

The `IC4Float4` class represents an AS/400 4-byte floating point number. For a `IC4Float4` object, `ExternalFormat` is a typedef of char[4]. The local format is `float`.

Derivation

```
IC4DataType
 IC4Float4
```

Header File

`ic4float.hpp`

Constructors

```
IC4Float4();
```

The default constructor initializes the local format data to zero.

```
IC4Float4(float value,
          ExternalFormat externalDataPointer =NULL);
IC4Float4(float* dataPointer,
          ExternalFormat externalDataPointer =NULL);
IC4Float4(const IC4Float4& other);
```

The parameters are the following:

value	(in) The initial value for the local format data.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.
dataPointer	(in) A pointer to where the local format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
other	(in) An `IC4Float4` object. The data item is copied by value.

Public Members

clone

```
virtual IC4DataType* clone() const;
```

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

dataPointter

```
virtual void* dataPointer() const;
```

Returns a pointer to the local format data.

Exceptions

- IC4DataTypeException

dataSize

```
virtual unsigned long dataSize() const;
```

Returns the size in bytes of the local format data.

externalDataPointer

```
virtual void* externalDataPointer() const;
```

Returns a pointer to the external format data.

Exceptions

- IC4DataTypeException

externalDataSize

```
virtual unsigned long externalDataSize() const;
```

Returns the size in bytes of the external format data.

operator=

```
IC4Float4& operator=(float value);
IC4Float4& operator=(const IC4Float4& other);
```

Assignment operators.

The parameters are the following:

value	(in) The value to assign to the local format data.
other	(in) An `IC4Float4` object from which to assign the local and the external format data values. The data item is copied by value.

operator float

```
operator float&();
operator float() const;
```

Conversion operators: Return a reference to the local format data or a copy of the local format data.

setDataPointer

```
void setDataPointer(float* dataPointer);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(ExternalFormat externalDataPointer);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
```

Converts the data from the external format to the local format.

translateToExternal

```
virtual void translateToExternal();
```

Converts the data to the external format from the local format.

Inherited Public Members

Member	Class	Page
codePageOverrideExpected	IC4DataType	144
overlay	IC4DataType	145

IC4Float8

The `IC4Float8` class represents an AS/400 8-byte floating point number. For an `IC4Float8` object, `ExternalFormat` is a typedef of char[8]. The local format is `double`.

Derivation

```
IC4DataType
  IC4Float8
```

Header File

`ic4float.hpp`

Constructors

```
IC4Float8();
```

The default constructor initializes the local format data to zero.

```
IC4Float8(double value,
          ExternalFormat externalDataPointer =NULL);
IC4Float8(double* dataPointer,
          ExternalFormat externalDataPointer =NULL);
IC4Float8(const IC4Float8& other);
```

The parameters are the following:

value	(in) The initial value for the local format data.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.
dataPointer	(in) A pointer to where the local format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
other	(in) An `IC4Float8` object. The data item is copied by value.

Public Members

clone

```
virtual IC4DataType* clone() const;
```

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

IC4Float8

dataPointer

```
virtual void* dataPointer() const;
```

Returns a pointer to the local format data.

Exceptions

- IC4DataTypeException

dataSize

```
virtual unsigned long dataSize() const;
```

Returns the size in bytes of the local format data.

externalDataPointer

```
virtual void* externalDataPointer() const;
```

Returns a pointer to the external format data.

Exceptions

- IC4DataTypeException

externalDataSize

```
virtual unsigned long externalDataSize() const;
```

Returns the size in bytes of the external format data.

operator=

```
IC4Float8& operator=(double value);
IC4Float8& operator=(const IC4Float8& other);
```

Assignment operators.

The parameters are the following:

value	(in) The value to assign to the local format data.
other	(in) An `IC4Float8` object from which to retrieve the local and external format data values. The data item is copied by value.

operator double

```
operator double&();
operator double() const;
```

Conversion operators: Return a reference to the local format data or a copy of the local format data.

setDataPointer

```
void setDataPointer(double* dataPointer);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(ExternalFormat externalDataPointer);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
```

Converts the data from the external format to the local format.

translateToExternal

```
virtual void translateToExternal();
```

Converts the data to the external format from the local format.

Inherited Public Members

Member	Class	Page
codePageOverrideExpected	IC4DataType	144
overlay	IC4DataType	145

IC4Struct

The `IC4Struct` class enables a group of data items to be manipulated as a single C++-type structure. A data item can be added to a structure only once, and can be added to only one structure. After a data item is added to a structure, it is called a member of the structure.

An `IC4Struct` object may have members that are dependent on character set code pages for proper conversion between AS/400 and PC formats, and these members may expect to have the code page values explicitly specified when conversion is done. See the base class, “IC4CPDDataType” on page 141, for additional information.

Derivation

```
IC4DataType
 IC4CPDDataType
  IC4Struct
```

Header File

```
ic4struc.hpp
```

Constructors

```
IC4Struct();
IC4Struct(void* dataPointer,
          unsigned long dataSize,
          void* externalDataPointer =NULL,
          unsigned long externalDataSize =0);
IC4Struct(const IC4Struct& other);
```

The parameters are the following:

dataPointer	(in) A pointer to where the local format data for the structure object is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
dataSize	(in) The size of the local format data pointed to by `dataPointer`. This value is the maximum size of the local format data for the `IC4Struct` object.
externalDataPointer	(in) A pointer to where the external format data for the structure is to be stored. Specifying NULL means that object-managed storage will be used unless you assign

	user-managed storage at a later time using the `setExternalDataPointer()` member function.
externalDataSize	(in) The size of the external format data pointed to by `externalDataPointer`. This value is the maximum size of the external format data for the `IC4Struct` object.
other	(in) An `IC4Struct` object that is to be copied. The structure is copied by value. The local and external format data of the newly constructed object will be in object-owned storage, not in user-managed storage.

Exceptions

- IC4DataTypeException

Public Members

addMember

`IC4Struct&` **`addMember`**`(IC4DataType&` *`member`*`);`

Adds a member to the `IC4Struct` object. Members must be added in the order that they are expected. Members cannot be added if this structure is a member of another `IC4Struct` object or is an element of an `IC4Array` object.

If user-managed storage is assigned to the structure, members are assigned storage locations and structure offsets as they are added. An exception is thrown if adding a member would cause the available space to be exceeded.

If user-managed storage is not assigned to the structure, then storage locations are not assigned to the members until another member function is called that requires the storage locations to be assigned. You must call one or more of the following member functions before accessing the storage location for the member directly:

`dataPointer()`	assigns storage for local format data
`externalDataPointer()`	assigns storage for external format data
`overlayAtOffsets()`	assigns storage for both local and external format data
`setDataPointer()`	assigns storage for local format data
`setExternalDataPointer()`	assigns storage for external format data
`translateFromExternal()`	assigns storage for both local and external format data
`translateToExternal()`	assigns storage for both local and external format data

The parameters are the following:

member	(in) An instance of an `IC4DataType` derived class. Must have a size greater than zero.

Exceptions

- IC4DataTypeException

IC4Struct

clone

`virtual IC4DataType* **clone**() const;`

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

dataPointer

`virtual void* **dataPointer**() const;`

Returns a pointer to the local format data. If user-managed storage has not been assigned for the local format data then:

- An exception is thrown if at least one member has not been added.
- No more members can be added after calling this member function.

Exceptions

- IC4DataTypeException

dataSize

`virtual unsigned long **dataSize**() const;`

Returns the size in bytes of the local format data.

externalDataPointer

`virtual void* **externalDataPointer**() const;`

Returns a pointer to the external format data. If user-managed storage has not been assigned for the external format data then:

- An exception is thrown if at least one member has not been added.
- No more members can be added after calling this member function.

Exceptions

- IC4DataTypeException

externalDataSize

`virtual unsigned long **externalDataSize**() const;`

Returns the size in bytes of the external format data.

operator<<

`IC4Struct& **operator<<**(IC4DataType& *member*);`

Adds a member to the `IC4Struct` object. Members must be added in the order that they are expected. and must have a size greater than zero. Members cannot be added if this structure is a member of another `IC4Struct` object or is an element of an `IC4Array` object.

If user-managed storage has been assigned to the structure, members are assigned storage locations and structure offsets as they are added. An exception is thrown if adding a member would cause the available space to be exceeded.

If user-managed storage is not assigned to the structure, then storage locations are not assigned to the members until another member function is called that requires the storage locations to be assigned. You must call one or more of the following member functions before accessing the data for the member:

`dataPointer()`	assigns storage for local format data
`externalDataPointer`	assigns storage for external format data
`overlayAtOffsets`	assigns storage for both local and external format data
`setDataPointer`	assigns storage for local format data
`setExternalDataPointer`	assigns storage for external format data
`translateFromExternal`	assigns storage for both local and external format data
`translateToExternal`	assigns storage for both local and external format data

The parameters are the following:

member	(in) An instance of an `IC4DataType` derived class. Must have a size greater than zero.

Exceptions

- IC4DataTypeException

overlayAtOffsets

```
void overlayAtOffsets(unsigned long dataOffset,
                      unsigned long externalDataOffset,
                      IC4DataType& dataItem);
```

Overlays this structure with a data item, allowing the same storage to be referred to by more than one data type (similar to a C++ union). This member function sets up the specified data item at the specified offsets in the structure's local format and external format data. If user-managed storage has not been assigned for the local format data and/or the external format data, then no more members can be added after calling this member function.

No bounds checking is performed other than verifying that the offsets are within the local and external format data. Therefore, a data item overlaying a structure may have data that extends beyond the structure's data.

The parameters are the following:

dataOffset	(in) The offset at which the local format data for the structure is to be overlaid. Value must be within the local format data for the structure.
externalDataOffset	(in) The offset at which the external format data for the structure is to be overlaid. Value must be within the external format data for the structure.

dataItem	(in) The data item that is to overlay the structure.

Exceptions

- IC4DataTypeException

setDataPointer

```
void setDataPointer(void* dataPointer,
                    unsigned long size =0);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage. If user-managed storage is assigned to hold the local format data and NULL is specified, then no more members can be added after this member function is called.
size	(in) The size in bytes of the user-managed storage being assigned. Must be greater than zero and large enough to contain the current size of the local format data. This parameter is ignored if *dataPointer* is NULL.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(void* externalDataPointer,
                            unsigned long size =0);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.

	If user-managed storage has been assigned to hold the external format data and NULL is specified, then no more members can be added after this member function is called.
size	(in) The size in bytes of the user-managed storage being assigned. Must be greater than zero and large enough to contain the current size of the external format data. This parameter is ignored if *externalDataPointer* is NULL.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
virtual void translateFromExternal(unsigned long codePage,
                                   unsigned long externalCodePage);
```

Converts the data from the external format to the local format. If code page values are specified, those values are used for members that return True for `codePageOverrideExpected()`. Otherwise, the code pages of the structure are used for members that return True for `codePageOverrideExpected`. If user-managed storage was not assigned for the local format data, then no more members can be added after calling this member function.

The parameters are the following:

codePage	(in) The code page to use for the local format data.
externalCodePage	(in) The code page to use for the external format data.

Exceptions

- IC4DataTypeException

translateToExternal

```
virtual void translateToExternal();
virtual void translateToExternal(unsigned long codePage,
                                 unsigned long externalCodePage);
```

Converts the data to the external format from the local format. If code page values are specified, those values are used for members that return True for `codePageOverrideExpected()`. Otherwise, the codes pages of the structure are used for members that return True for `codePageOverrideExpected()`. If user-managed storage was not assigned for the external format data, then no more members can be added after calling this member function.

The parameters are the following:

codePage	(in) The code page to use for the local format data.
externalCodePage	(in) The code page to use for the external format data.

Exceptions

- IC4DataTypeException

Inherited Public Members

Member	Class	Page
codePage	IC4CPDDataType	141
codePageOverrideExpected	IC4CPDDataType	141
externalCodePage	IC4CPDDataType	142
overlay	IC4DataType	145
resetCodePages	IC4CPDDataType	142
setCodePage	IC4CPDDataType	142
setCodePageOverrideExpected	IC4CPDDataType	142
setCodePages	IC4CPDDataType	142
setExternalCodePage	IC4CPDDataType	143

IC4UnsignedBin2

The `IC4UnsignedBin2` class represents an AS/400 unsigned 2-byte binary number. For an `IC4UnsignedBin2` object, `ExternalFormat` is a typedef of char[2]. The local format is `unsigned short int`.

Derivation

```
IC4DataType
 IC4UnsignedBin2
```

Header File

```
ic4bin.hpp
```

Constructors

```
IC4UnsignedBin2();
```

The default constructor initializes the local format data to zero.

```
IC4UnsignedBin2(unsigned short value,
                ExternalFormat externalDataPointer =NULL);
IC4UnsignedBin2(unsigned short* dataPointer,
                ExternalFormat externalDataPointer =NULL);
IC4UnsignedBin2(const IC4UnsignedBin2& other);
```

The parameters are the following:

value	(in) The initial value for the local format data.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.

dataPointer	(in) A pointer to where the local format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
other	(in) An `IC4UnsignedBin2` object.

Public Members

clone

`virtual IC4DataType* **clone**() const;`

Returns a pointer to a new copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

dataPointer

`virtual void* **dataPointer**() const;`

Returns a pointer to the local format data.

Exceptions

- IC4DataTypeException

dataSize

`virtual unsigned long **dataSize**() const;`

Returns the size in bytes of the local format data.

externalDataPointer

`virtual void* **externalDataPointer**() const;`

Returns a pointer to the external format data.

Exceptions

- IC4DataTypeException

externalDataSize

`virtual unsigned long **externalDataSize**() const;`

Returns the size in bytes of the external format data.

operator short

```
operator unsigned short&();
operator unsigned short() const;
```

Conversion operators: Return a reference to the local format data or a copy of the local format data.

IC4UnsignedBin2

operator=

```
IC4UnsignedBin2& operator=(unsigned short value);
IC4UnsignedBin2& operator=(const IC4UnsignedBin2& other);
```

Assignment operators.

The parameters are the following:

value	(in) The value to assign to the local format data.
other	(in) An `IC4UnsignedBin2` object from which to retrieve the local and external format data values.

setDataPointer

```
void setDataPointer(unsigned short* dataPointer);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(ExternalFormat externalDataPointer);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
```

Converts the data from the external format to the local format.

translateToExternal

```
virtual void translateToExternal();
```

Converts the data to the external format from the local format.

Inherited Public Members

Member	Class	Page
codePageOverrideExpected	IC4DataType	144
overlay	IC4DataType	145

IC4UnsignedBin4

The `IC4UnsignedBin4` class represents an AS/400 unsigned 4-byte binary number. For an `IC4UnsignedBin4` object, `ExternalFormat` is a typedef of char[4]. The local format is `unsigned long int`.

Derivation

```
IC4DataType
 IC4UnsignedBin4
```

Header File

```
ic4bin.hpp
```

Constructors

```
IC4UnsignedBin4();
```

The default constructor initializes the local format data to zero.

```
IC4UnsignedBin4(unsigned long value,
                ExternalFormat externalDataPointer =NULL);
IC4UnsignedBin4(unsigned long* dataPointer,
                ExternalFormat externalDataPointer =NULL);
IC4UnsignedBin4(const IC4UnsignedBin4& other);
```

The parameters are the following:

value	(in) The initial value for the local format data.
externalDataPointer	(in) A pointer to where the external format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setExternalDataPointer()` member function.

dataPointer	(in) A pointer to where the local format data is to be stored. Specifying NULL means that object-managed storage will be used unless you assign user-managed storage at a later time using the `setDataPointer()` member function.
other	(in) An `IC4UnsignedBin4` object.

Public Members

clone

```
virtual IC4DataType* clone() const;
```

Returns a pointer to a copy of the data item. The caller is responsible for deleting the returned data item when it is no longer needed.

dataPointer

```
virtual void* dataPointer() const;
```

Returns a pointer to the local format data.

Exceptions

- IC4DataTypeException

dataSize

```
virtual unsigned long dataSize() const;
```

Returns the size in bytes of the local format data.

externalDataPointer

```
virtual void* externalDataPointer() const;
```

Returns a pointer to the external format data.

Exceptions

- IC4DataTypeException

externalDataSize

```
virtual unsigned long externalDataSize() const;
```

Returns the size in bytes of the external format data.

operator=

```
IC4UnsignedBin4& operator=(unsigned long value);
IC4UnsignedBin4& operator=(const IC4UnsignedBin4& other);
```

Assignment operators.

The parameters are the following:

value	(in) The value to assign to the local format data.
other	(in) An `IC4UnsignedBin4` object from which to retrieve the local and external format data values.

operator unsigned long

```
operator unsigned long&();
operator unsigned long() const;
```

Conversion operators: Return a reference to the local format data or a copy of the local format data.

setDataPointer

```
void setDataPointer(unsigned long* dataPointer);
```

Assigns user-managed storage for the local format data.

The parameters are the following:

dataPointer	(in) A pointer to where the local format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the local format data is preserved in object managed storage. If a value other than NULL is specified, the existing value of the local format data is lost and the local format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

setExternalDataPointer

```
void setExternalDataPointer(ExternalFormat externalDataPointer);
```

Assigns user-managed storage for the external format data.

The parameters are the following:

externalDataPointer	(in) A pointer to where the external format data is to be stored. Storage can be unassigned by passing NULL. When storage is unassigned, the current value of the external format data is preserved in object-managed storage. If a value other than NULL is specified, the existing value of the external format data is lost and the external format data assumes the value in the newly assigned user-managed storage.

Exceptions

- IC4DataTypeException

translateFromExternal

```
virtual void translateFromExternal();
```

Converts the data from the external format to the local format.

translateToExternal

```
virtual void translateToExternal();
```

Converts the data to the external format from the local format.

Inherited Public Members

Member	Class	Page
codePageOverrideExpected	IC4DataType	144
overlay	IC4DataType	145

Chapter 7. User Space Class

IC4UserSpace

The `IC4UserSpace` class represents an AS/400 user space object.

Derivation

`IC4UserSpace` does not derive from another class.

Header File

`ic4us.hpp`

Constructors

```
IC4UserSpace(const char* userSpaceName,
             const char* systemName =NULL);
```

```
IC4UserSpace(const IC4UserSpace& userSpace);
```

The parameters are the following:

userSpaceName	(in) The qualified name of the user space specified as a path name in the library file system. For example, `/QSYS.LIB/MYLIB.LIB/MYUS.USRSPC`. The library value can be one of the following: • `a library name` • `%CURLIB%` • `%LIBL%`
systemName	(in) The name of the AS/400 system on which the user space is located. This is an optional parameter. If not specified, the default system name for Client Access/400 is used.
userSpace	(in) An `IC4UserSpace` object. If this object is open, then the newly constructed object is opened.

Exceptions

- IC4UserSpaceException

Public Members

automaticExtendibility

```
Extendibility automaticExtendibility() const;
```

Returns `NotExtendible` if the user space is not automatically extendible and `Extendible` if the user space is automatically extendible.

Exceptions

- IC4UserSpaceException

changeAutomaticExtendibility

```
void changeAutomaticExtendibility(Extendibility extendibility);
```

Changes the value of the attribute that determines whether a user space can be extended automatically by the system.

The parameters are the following:

extendibility	(in) The new value for the attribute. For information on *extendibility*, see "Extendibility" on page 171.

Exceptions

- IC4UserSpaceException

changeInitialValue

```
void changeInitialValue(char newValue);
```

Changes the value of the attribute that indicates the value used to initialize future extensions of the user space.

The parameter is the following:

newValue	(in) The new value for the attribute. Any character is valid. For best performance use '\0'.

Exceptions

- IC4UserSpaceException

changeSize

```
void changeSize(unsigned long newSize);
```

Changes the size of the user space.

The parameter is the following:

newSize	(in) The new size of the user space in bytes. Allowable values are 1 to 16,776,704 bytes. If the new size is smaller than the previous size, the user space is truncated. If the new size is larger than the previous size, the user space is extended and initialized as specified by the initial value attribute.

Exceptions

- IC4UserSpaceException

close

```
void close();
```

Closes the communications conversation that was started with `open()`.

Exceptions

- IC4UserSpaceException

create

```
void create(const char* domain,
             unsigned long length,
             const char* replaceUserSpace =NULL,
             const char* extendedAttribute =NULL,
             char initialValue ='\0',
             const char* textDescription =NULL,
             const char* authority =NULL);

void create(unsigned long length,
             const char* replaceUserSpace =NULL,
             const char* extendedAttribute =NULL,
             char initialValue ='\0',
             const char* textDescription =NULL,
             const char* authority =NULL;
```

Creates a user space on the AS/400. The user space can be created in a specific library or in the current library.

The parameters are the following:

domain	(in) The domain in which the user space is created. This is an optional parameter that defaults to *SYSTEM. Other possible values are: • `*DEFAULT` • `*USER` For information about the security aspects of user space domains, see *System API Reference*.
length	(in) The maximum length of the user space in bytes. Allowable values are 1 to 16,776,704 bytes.
replaceUserSpace	(in) Specifies whether the user space should be replaced if it already exists. This is an optional parameter that defaults to *NO. The other possible value is *YES.
extendedAttribute	(in) The extended attribute with which the user space should be created. This is an optional, user-defined value that can be used as an identifier for the user space. If not specified, the extended attribute is set to blank.
initialValue	(in) The initial value given to the user space when it is created. This is an optional parameter and the default value is hexadecimal zero for optimum performance.

textDescription	(in) Description of the user space. This is an optional parameter. The default value is a blank string.
authority	(in) The public access authority given to users who do not have specific authority to the user space. This is an optional parameter that defaults to `*LIBCRTAUT`. Other possible values are: • `*CHANGE` • `*ALL` • `*USE` • `*EXCLUDE` • `authorization list name`

Exceptions

- IC4UserSpaceException

destroy

`void` **`destroy`**`();`

Deletes the user space from the AS/400 system, and does an implicit close() of the user space object.

Exceptions

- IC4UserSpaceException

initialValue

`char` **`initialValue`**`() const;`

Returns the initial value of the user space when it is created or when the initial value was set using `changeInitialValue()`.

Exceptions

- IC4UserSpaceException

isOpen

`IBoolean` **`isOpen`**`() const;`

Returns True if the user space is open.

name

`const IString&` **`name`**`() const;`

Returns the fully qualified name of the user space specified on the constructor.

open

`void` **`open`**`();`

Starts a conversation with the AS/400 system specified on the constructor and validates that the user space exists on the system.

Exceptions

- IC4UserSpaceException

operator=

```
IC4UserSpace& operator=(const IC4UserSpace& userSpace);
```

Assignment operator: If the user space being copied is open, then this user space is open when the assignment operation is complete.

Exceptions

- IC4UserSpaceException

operator==

```
IBoolean operator==(const IC4UserSpace& userSpace) const;
```

Equality operator: returns True if the user space name and system name are equal.

Exceptions

- IC4UserSpaceException

operator!=

```
IBoolean operator!=(const IC4UserSpace& userSpace) const;
```

Inequality operator: returns True if the user space name and system name are not equal.

Exceptions

- IC4UserSpaceException

operator<

```
IBoolean operator<(const IC4UserSpace& userSpace) const;
```

Less-than operator: returns True if the left operand is less than the right operand as determined by the alphabetical order of the system name and then the qualified user space name.

Exceptions

- IC4UserSpaceException

operator>

```
IBoolean operator>(const IC4UserSpace& userSpace) const;
```

Greater-than operator: returns True if the left operand is greater than the right operand as determined by the alphabetical order of the system name and then the qualified user space name.

Exceptions

- IC4UserSpaceException

read

```
void read(IString& readData,
          unsigned long readLength =1,
          unsigned long startingPosition =1);
void read(char* readData,
          unsigned long readLength,
          unsigned long startingPosition =1);
```

Reads the contents of the user space for the specified length.

The parameters are the following:

readData	(out) String to receive the contents of the user space.
readLength	(in) The number of bytes to read from the user space.
startingPosition	(in) The 1-based position at which to start reading the user space. This is an optional parameter with a default value of 1.

Exceptions

- IC4UserSpaceException

size

```
unsigned long size() const;
```

Returns the size of the user space in bytes.

Exceptions

- IC4UserSpaceException

systemName

```
const IString& systemName() const;
```

Returns the name of the AS/400 system on which the user space is located. This is an optional parameter. If not specified, the default system name for Client Access is used.

Exceptions

- IC4UserSpaceException

write

```
void write(const char* writeData,
           unsigned long writeDataLength =0,
           unsigned long startingPosition =1,
           Force force =NoAuxiliaryForce);
```

Writes data to the user space.

The parameters are the following:

writeData	(in) The value to write to the user space.
writeDataLength	(in) The length of the data to write. If this value is 0, *writeData* must be null-terminated.
startingPosition	(in) The position in the user space at which to begin writing the new data.
force	(in) Indicates whether the data should be forced to auxiliary storage. This is an optional parameter. For information on *force*, see “Force.”

Exceptions

- IC4UserSpaceException

Enumerations

The following describes the Extendibility enumeration and the Force enumeration.

Extendibility

```
enum Extendibility { NotExtendible =0, Extendible };
```

Use these values for the user space attribute that specifies whether the user space is automatically extended when the end of the user space is encountered.

Force

```
enum Force { NoAuxiliaryForce =0,
             AsynchronousForce,
             SynchronousForce
             };
```

Use these values for the user space attribute that specifies whether the user space is forced to auxiliary storage whenever the value of the user space is changed.

IC4UserSpace

Appendix A. Classes and Header Files

The following table lists each Access Class Library class and the header files associated with the classes.

Table 5 (Page 1 of 2). Classes and Their Respective Header Files

Class	Header File
IC4BaseDataQueue	IC4BASDQ.HPP
IC4DataQueue	IC4DQ.HPP
IC4KeyedDataQueue	IC4KEYDQ.HPP
IC4DataQueueException	IC4DQEXC.HPP
IC4CharacterDataArea	IC4CDA.HPP
IC4BaseDataArea	IC4BDA.HPP
IC4DecimalDataArea	IC4DDA.HPP
IC4LogicalDataArea	IC4LDA.HPP
IC4DataAreaException	IC4DAEXC.HPP
IC4SQLDatabase	IC4SQDB.HPP
IC4SQLEnvironment	IC4SQENV.HPP
IC4SQLRow	IC4SQROW.HPP
IC4SQLResultSet	IC4SQRS.HPP
IC4SQLStatement	IC4SQST.HPP
IC4SQLVariable	IC4SQVA.HPP
IC4SQLVariableList	IC4SQVL.HPP
IC4SQLException	IC4SQEXC.HPP
IC4UserSpace	IC4US.HPP
IC4UserSpaceException	IC4USEXC.HPP
IC4CommandProcessor	IC4CP.HPP
IC4CommandProcessorException	IC4CPEXC.HPP
IC4OS400Command	IC44CMD.HPP
IC4Command	IC4CMD.HPP
IC4CommandException	IC4CMEXC.HPP
IC4Exception	IC4EXCPT.HPP
IC4Translate	IC4XLT.HPP
IC4TranslateException	IC4XLEXC.HPP
IC4ProgramException	IC4PGEXC.HPP
IC4Program	IC4PGM.HPP

Classes and Header Files

Table 5 (Page 2 of 2). Classes and Their Respective Header Files

Class	**Header File**
IC4Array IC4ArrayBase	IC4ARRAY.HPP
IC4Bin2 IC4Bin4 IC4UnsignedBin2 IC4UnsignedBin4	IC4BIN.HPP
IC4Byte	IC4BYTE.HPP
IC4Char	IC4CHAR.HPP
IC4DataType	IC4DT.HPP
IC4CPDDataType	IC4DTCPD.HPP
IC4DataTypeException	IC4DTEXC.HPP
IC4Float4 IC4Float8	IC4FLOAT.HPP
IC4Struct	IC4STRUC.HPP
<Not a class>	IC4DT400.HPP
<Not a class>	IC4SQDEF.H

Bibliography

This bibliography lists related publications that you may find helpful:

- The *Client Access/400 for DOS and OS/2 API Reference*, SC41-3562, provides detailed information about the application program interfaces (APIs) provided with Client Access for DOS and OS/2.
- The *Client Access for Windows 95 API and Technical Reference*, SC41-3513, contains the application programming interface information needed to write cooperative processing applications for Client Access/400.
- The *Client Access/400 Optimized for OS/2 API and Technical Reference*, SC41-3511, provides information about using the Client Access/400 Optimized for OS/2 API interface to create applications. Includes a description of the structure of API calls and examples of calls in certain programming languages.
- The *CL Reference*, SC41-4722, provides a description of the AS/400 control language (CL) and its OS/400 commands. (Non-OS/400 commands are described in the respective licensed program publications.) Also provides an overview of *all* the CL commands for the AS/400 system, and it describes the syntax rules needed to code them.
- The *IBM Access Class Library User's Guide*, SC41-4623, provides information on how to use the access class library to access AS/400 data and services when using VisualAge for C++ This book discusses accessing data and services from OS/2, OS/2 Optimized, OS/400, Windows 95, and Windows NT. The following C++ classes are covered: command, data area, database, data queue, data translation, data types, program, and user space.
- The *IBM Access Class Library for OS/400 Reference*, SC41-4620, provides reference information on the access class library for the OS/400 server. This book helps you create VisualAge for C++ for AS/400 applications that run on the AS/400 server and access AS/400 data and services.
- The *ILE COBOL/400 Programmer's Guide*, SC09-2072, provides information about how to write, compile, bind, run, debug, and maintain ILE COBOL/400 programs on the AS/400 system. It provides programming information on how to call other ILE COBOL/400 and non-ILE COBOL/400 programs, share data with other programs, use pointers, and handle exceptions. It also describes how to perform input/output operations on externally attached devices, database files, display files, and ICF files.
- The *ILE Concepts*, SC41-4606, explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- The *ILE RPG/400 Programmer's Guide*, SC09-2074, provides information about the ILE RPG/400 programming language, which is an implementation of the RPG IV language in the Integrated Language Environment (ILE) on the AS/400 system. It includes information on creating and running programs, with considerations for procedure calls and interlanguage programming. The guide also covers debugging and exception handling and explains how to use AS/400 files and devices in RPG programs. Appendixes include information on migration to RPG IV and sample compiler listings. It is intended for people with a basic understanding of data processing concepts and of the RPG language.
- The *Integrated File System Introduction*, SC41-4711, provides an overview of the integrated file system, which includes:
  - What it is
  - Why you might want to use it
  - Concepts and terminology
  - The interfaces you can use to interact with it
  - The APIs and techniques you can use in programs that interact with it
  - Characteristics of individual file systems
- The *System API Reference*, SC41-4801, provides information for the experienced programmer on how to use the application programming interfaces (APIs) to such OS/400 functions as:
  - Dynamic Screen Manager
  - Files (database, spooled, hierarchical)
  - Message handling
  - National language support
  - Network management
  - Objects
  - Problem management
  - Registration facility

  - Security
  - Software products
  - Source debug
  - UNIX**-type
  - User-defined communications
  - User interface
  - Work management

  Includes original program model (OPM), Integrated Language Environment (ILE), and UNIX-type APIs.

- The *VisualAge for C++ for AS/400 C Library Reference*, SC09-2441, describes the run-time library functions available to C++ programs developed with VisualAge for C++ for AS/400.
- The *VisualAge for C++ for AS/400 C++ Language Reference*, SC09-2442, describes the C++ language as implemented by VisualAge for C++ for AS/400, including keywords, preprocessor directives, and constructs.
- The *VisualAge for C++ for AS/400 C++ Programming Guide*, SC09-2417, describes programming techniques to write ILE C++ applications that run on AS/400. Topics include performing I/O operations; working with AS/400 files, devices, pointers, and locales; implementing interlanguage calls; using templates; handling exceptions; and porting code.
- The *VisualAge for C++ for AS/400 C++ User's Guide*, SC09-2416, explains how to compile, bind, and debug C++ applications that run in the Integrated Language Environment on the AS/400.
- The *VisualAge for C++ for AS/400 IBM Open Class Library Reference*, SC09-2440, describes the classes from the IBM Open Class Library: I/O Streams, Complex Mathematics, Collections, Data Types and Exceptions. Also describes the Binary Coded Decimal Class Library for OS/400.
- The *VisualAge for C++ for AS/400 IBM Open Class Library User's Guide*, SC09-2443, shows how to create C++ programs with the many different classes of the IBM Open Class Library: I/O Streams, Complex Mathematics, Collections, Data Types and Exceptions, and Database Access. Also describes how to create C++ programs using the IBM Binary Coded Decimal Class Library for OS/400.

The following book contains additional information that you may find helpful:

- *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*, ISBN 1-55615-658-8

Index

Reader Comments—We'd Like to Hear from You!

AS/400 Advanced Series
IBM Access Class Library
for Windows Reference**
Version 3

Publication No. SC41-4622-00

Overall, how would you rate this manual?

	Very Satisfied	Satisfied	Dissatisfied	Very Dissatisfied
Overall satisfaction				

How satisfied are you that the information in this manual is:

Accurate				
Complete				
Easy to find				
Easy to understand				
Well organized				
Applicable to your tasks				
T H A N K Y O U !				

Please tell us how we can improve this manual:

May we contact you to discuss your responses? __ Yes __ No
Phone: (____) ___________ Fax: (____) ___________ Internet: ___________

To return this form:

- Mail it
- Fax it
  United States and Canada: **800+937-3430**
  Other countries: **(+1)+507+253-5192**
- Hand it to your IBM representative.

Note that IBM may use or distribute the responses to this form without obligation.

Name

Address

Company or Organization

Phone No.

Reader Comments—We'd Like to Hear from You!
SC41-4622-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN DEPT 542 IDCLERK
IBM CORPORATION
3605 HWY 52 N
ROCHESTER MN 55901-9986

Fold and Tape **Please do not staple** Fold and Tape

SC41-4622-00

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC41-4622-00

Spine information:

IBM

AS/400 Advanced Series

IBM Access Class Library
for Windows** Reference

Version 3