

Raima Database Manager 11.0

C++ Interface Users Guide

Trademarks

Raima Database Manager® (RDM®), RDM Embedded® and RDM Server® are trademarks of Raima Inc. and may be registered in the United States of America and/or other countries. All other names may be trademarks of their respective owners.

This guide may contain links to third-party Web sites that are not under the control of Raima Inc. and Raima Inc. is not responsible for the content on any linked site. If you access a third-party Web site mentioned in this guide, you do so at your own risk. Inclusion of any links does not imply Raima Inc. endorsement or acceptance of the content of those third-party sites.

Contents

Contents	3
Introduction	5
Interfaces	5
Db Interface	5
Cursor Interface	6
Creating the C++ Interface	7
Access Methods	8
Singleton Cursors	8
Table Scan Cursors	8
Key Scan Cursors	8
Set Member Cursor	8
Operational Overview	9
Database Control Methods	9
Opening	9
Initializing	9
Destroy	9
Close	10
Transaction Control Methods	11
Update Transactions	11
Read Transactions	13
Database Snapshots	14
Transaction Commits	15
Transaction Aborts	16
Precommit	16
New Methods	17
Delete Methods	19
Get Methods	20

Set Methods	21
Blob Methods	22
Network Model Set Methods	24
Cursor Creation Methods	27
Table Scan Cursors	27
Key Scan Cursors	29
Set Member Cursors	32
Singleton Cursors	34
Special Singletons	35
Cursor Navigation Methods	36
First	36
Last	37
Next	37
Prev	38
Count	39
Cursor Comparison Methods	40
Equal	40
Not Equal	40
Less Than	41
Less Than Or Equal	41
Greater Than	41
Greater Than Or Equal	42
Error Handling	43
Exceptions	43
Glossary	44
Index	55

Introduction

The RDM database management system (DBMS) is designed to provide powerful, flexible, high-performance capabilities for developing embedded database applications. By combining the network and relational model technologies in a single system, RDM lets you organize and access information efficiently, regardless of the complexity of the data. The Native RDM C API provides an efficient low-level interface with a rich set of functionality.

The C++ interface to RDM is designed to augment the RDM navigation API by providing database specific sets of classes implementing higher-level abstractions of the Core API. These generated interfaces are built to fit your specific database schema using an object orientated approach. The C++ API is designed for ease of use, but by giving full access to both RDM's network and relational functionality for RDM, it is very powerful and can be used to create efficient database applications.

Interfaces

There are two main interfaces that comprise the RDM C++ API, the Db interface which encapsulates access to a particular database and the Cursor interface which encapsulates access to records within a database. These interfaces contain methods that are common to all databases and records and methods that are specific to a particular schema. By using these interfaces the C++ programmer is able to create applications that safely and efficiently query, insert, update, and delete data stored in an RDM database

The interfaces are implemented as smart pointers (http://en.wikipedia.org/wiki/Smart_pointer) to implementation objects. Therefore the interface objects should be passed by value not reference. There are explicit public methods for the default constructor, copy constructor, assignment operator, and destructor that implement the smart pointer. The developer only needs to know that whenever the last interface reference goes out of scope the database will automatically be closed and the implementation object will be de-allocated.

Db Interface

There is one Db interface class generated for each database access through the RDM C++ API. This database specific Db interface extends the base Db class and provides methods that allow the programmer to:

- Open the database
- Create a new records
- Obtain record scan cursors
- Obtain key scan cursors
- Start read transactions
- Start write transactions
- Pre-commit transaction
- Commit transactions
- Abort transactions
- Start a read-only snap shot

Cursor Interface

There is one Cursor interface class generated for each record defined in a schema. These Cursor classes extend the base Cursor class and provide methods that allow the programmer to:

- Delete Records
- Disconnect Records from a set owner
- Connect Records to a set owner
- Reconnect Records to a new set owner
- Navigate through the records in the cursor
- Obtain record scan cursors
- Obtain a key scan cursor
- Obtain a cursor containing member records
- Obtain a singleton cursor containing an owner record
- Update individual field values
- Update all field values
- Read individual field values
- Read all field values
- Find records by a key value
- Obtain a singleton cursor

Creating the C++ Interface

The C++ API is tailored to a specific schema and is optionally generated by the Data Definition Language Process (ddlp) utility when it processes a database schema definition file. As an example we will create a schema to support a hypothetical piece of equipment that stores readings from onboard sensors. Information on how to design a RDM schema can be found in [section 4.2](#) of the RDM User's Guide.

```
database measurements
{
  data file "measurements.d00" contains sensor;
  data file "measurements.d01" contains measurement;
  key file  "measurements.k00" contains sensor.name;
  blob file "measurements.b00" contains measurement.raw_data;

  record sensor {
    unique key char name[32];
    int32_t status;
  }

  record measurement {
    int32_t time;
    int32_t value;
    blob_id raw_data;
  }

  set sensor_measurement {
    order last;
    owner sensor;
    member measurement;
  }
}
```

We run the DDLP utility with the `-cpp` option to get a C++ API for the above schema.

```
ddlp -d -cpp measurement.ddl
```

This command will produce the source and header that comprise the RDM C++ API for the measurements database

<code>measurements.h</code>	RDM database header file
<code>measurements_api.h</code>	RDM C++ interface header file
<code>measurements_api.cpp</code>	RDM C++ interface implementation file

All source files that want to use the RDM C++ API for the measurements database will need to include the `measurements_api.h` header file. The `measurements_api.cpp` source file will need to be compiled with the application code.

Access Methods

Cursors are used to navigate, read, update, create, and delete records within the C++ API. There are four main types of Cursors available each of which implements a different access method to data stored in a RDM database. While there are four types of Cursor each cursor implementation uses the exact same interface which simplifies the uses of these cursors.

Singleton Cursors

Singleton Cursors contain one and only one record instance. Singleton Cursors can be obtained by

- Creating a new record (`Db.New_recordname`)
- Asking for the owner of a set (`Cursor.Get_setname_owner`)
- Asking for a Singleton cursor from any cursor (`Cursor.GetThis`)

Singleton cursors can be useful if we want to save a reference to a particular record or if we wish to do multiple things to with the record and an operation may remove the record from the cursor.

Table Scan Cursors

Table scan cursors contain a collection of all the records in a table in no particular order. Table Scan Cursors are useful when you want to access all records in a table and the order of access is not important. For example if you wanted to calculate the total number of measurements taken by all sensors in our example database. Table Scan Cursors can be obtained from the Db Interface by the `Db->Get_recordname_records` method and from a Cursor interface by using the `Cursor->GetRecords` method. Obtaining a Cursor from the Db interface positions the cursor at the first cursor position. Obtaining a Cursor from another Cursor positions the new cursor at the same Cursor position.

Key Scan Cursors

Key scan cursors contain a collection of all records in a table in index order. Key Scan Cursor can be useful when you want to display records to a user in a particular order. Key Scan Cursors can be obtained from the Db Interface by the `Db.Get_recordname_recordsBy_indexname` method or from a Cursor interface by the `Cursor.GetRecordsBy_indexname` method. Obtaining a Cursor from the Db interface positions the cursor at the first cursor position. Obtaining a Cursor from another Cursor positions the new cursor at the same Cursor position.

Set Member Cursor

A Set Member Cursor contains a collection of records associated with a particular owner record. Set Member Cursors can be used to iterate through all of the measurements made by a particular sensor in our example database. Set Member Cursors can only be obtained from a Cursor by the `Cursor.Get_setname_members` and `Cursor.Get_setname_siblings` methods.

Operational Overview

The C++ API allows developers to create powerful embedded applications without needing to worry about the low-level details required by the RDM Navigation APIs. While the C++ API contains a large set of functionality it does not implement everything that is supported by the RDM Core API. In this sections we look at the operations that are supported by the C++ API.

Database Control Methods

There are methods implemented in the Db interface that allow the developer to Open, Create, Destroy and Initialize the database our C++ API classes encapsulate.

Opening

A database is opened by using one of the overloaded static open methods generated in the Db interface. Calling these static methods give the application the database specific Db interface that is used to access the RDM database throughout the application.

```
static Db_measurements Open (...) throw (rdm_exception&);
```

This method has two optional

name	The database name to use for the open call. The default value for this parameter is the name defined in the schema.
mode	The mode to open the database in. The default is multi-user shared mode ("s")

Initializing

A database can be initialized by using the Initialize method in the Db interface. The Initialize method will (re) create all data, key, index, and blob files defined in the database schema. Any existing data stored in the database will be lost when calling this method.

```
void Initialize (void) const throw (rdm_exception&);
```

Destroy

A database can be destroyed by using the Destroy method in the Db interface. The Destroy method will remove all data, key, index, blob, log, and dbd files that make up a database. All Db and Cursor interfaces referencing this database become invalid after calling Destroy. All data stored in the database is lost when calling this method.

```
void Destroy (void) const throw (rdm_exception&);
```

Close

There is no explicit method for closing a database. The smart pointer implementation will automatically close the database when all references to the Db interface are de-allocated, reassigned, explicitly released or go out of scope. When this happens all internal resources held for the Db interface will be freed.

The following example demonstrates the usage of Open, Initialize and Destroy.

```
/* Measurement C++ API example application */
int32_t EXTERNAL_FCN measurements_main(
    int32_t      argc,
    const char *const *argv)
{
    Db_measurements db;

    UNREF_PARM(argc)
    UNREF_PARM(argv)

    try
    {
        /* Open the measurement database */
        db = db.Open("s");

        /* Remove any existing data from the database */
        db.Initialize();
    }
    catch(rdm_exception& e)
    {
        cerr << "Error " << e.error() << " opening/initializing database: " <<
e.what();
        return 1;
    }

    /* Perform operations with db */
    . . .

    /* Remove all database files before cleanly exiting */
    try
    {
        db.Destroy();
    }
    catch(rdm_exception& e)
    {
        cerr << "Error " << e.error() << " destroying database: " << e.what();
        return 1;
    }
}
```

```
    return 0;
}
```

Transaction Control Methods

The Db interface contains the methods providing transaction management functionality for the C++ API. When using the C++ API all read and write operations are executed within the context of a "read" or "write" transaction. This differs slightly from the convention of the core API where transactions are only required for "write" operations. The C++ API allows for simultaneous read transactions, but requires exclusive access to records for write transactions. This is similar to the core API except that in the C++ API lock management is encapsulated within transaction handling.

Update Transactions

All operations which may modify the contents of a database must be performed within the context of a "write" transaction. While the C++ interface will implicitly begin and end write transactions for the developer, it is considered poor programming practice to rely on these implicit transactions. Instead developers should explicitly start write transactions by using the `Db.BeginUpdate` method.

```
void BeginUpdate () const throw (rdm_exception&);
```

This method has two optional parameters

types	An array of recordType Identifiers to be locked
num	The number of recordType values in the array

If you do not provide parameters for the `BeginUpdate` method every record type and key file in the database will be write locked. In the following example both the measurement and sensor records will be locked by the Update method.

```
void add_measurement (
    Db_measurements db,
    struct measurement *sensor_fields,
    Cursor_sensor s) throw (rdm_exception&)
{
    Cursor_measurement m;
    db.BeginUpdate ();
    try
    {
        m = db.New_measurement_recordWithFieldValues (sensor_fields);
        m.Connect_sensor_measurement (s);
        db.End ();
    }
    catch (rdm_exception e)
```

C++ Interface Users Guide

```
{
    db.Abort();
}
}
```

If you only need to lock certain records for your operation you can specify those records using the `BeginUpdate` parameters.

```
Cursor_sensor add_sensor(
    Db_measurements db,
    struct sensor *sensor_fields) throw (rdm_exception&)
{
    Cursor_sensor s;
    recordType rtypes[] = {Cursor_sensor::Type()};
    db.BeginUpdate(rtypes, 1);
    try
    {
        s = db.New_sensor_recordWithFieldValues(sensor_fields);
        db.End();
    }
    catch (rdm_exception e)
    {
        db.Abort();
    }
    return s;
}
```

You can also use a special end marker in the array instead of specifying the number of record types.

```
Cursor_sensor add_sensor2(
    Db_measurements db,
    struct sensor *sensor_fields) throw (rdm_exception&)
{
    Cursor_sensor s;
    recordType rtypes[] = {Cursor_sensor::Type(), Cursor::NoMoreTypes()};
    db.BeginUpdate(rtypes);
    try
    {
        s = db.New_sensor_recordWithFieldValues(sensor_fields);
        db.End();
    }
    catch (rdm_exception e)
    {
        db.Abort();
    }
    return s;
}
```

Read Transactions

Read Transactions allow an application to read data from a database without blocking other readers. While it is legal to read data using Update Transactions those require exclusive access to the records and therefore do not promote parallelism to the same degree. The C++ interface will implicitly begin and end read transactions, but it is considered better programming to explicitly begin and end transactions.

```
void BeginRead () const throw (rdm_exception&);
```

If you do not provide parameters for the `BeginRead` method every record type and key file in the database will be read locked. In the following example both the measurement and sensor records will be locked by the `BeginRead` method.

```
void read_measurement(
    Db_measurements    db,
    Cursor_measurement m,
    struct sensor      *sensor_fields,
    struct measurement *measurement_fields) throw (rdm_exception&)
{
    db.BeginRead();
    try
    {
        m.GetFieldValues(measurement_fields);
        m.Get_sensor_measurement_owner().Get_name(sensor_fields->name);
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

If you only need to lock certain records for your operation you can specify those records using the `BeginRead` parameters.

```
void read_sensor(
    Db_measurements db,
    Cursor_sensor   s,
    struct sensor   *sensor_fields) throw (rdm_exception&)
{
    recordType rtypes[]= {Cursor_sensor::Type()};
    db.BeginRead(rtypes, 1);
    try
    {
        s.GetFieldValues(sensor_fields);
    }
}
```

```
catch (rdm_exception e)
{
    db.End();
    throw e;
}
db.End();
}
```

You can also use a special end marker in the array instead of specifying the number of record types.

```
void read_sensor2(
    Db_measurements db,
    Cursor_sensor s,
    struct sensor *sensor_fields) throw (rdm_exception&)
{
    recordType rtypes[] = {Cursor_sensor::Type(), Cursor::NoMoreTypes()};
    db.BeginRead(rtypes);
    try
    {
        s.GetFieldValues(sensor_fields);
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Database Snapshots

While read transactions allow for multiple readers they block all potential writers. If you would like to read data from the database without blocking writers you can utilize database snapshots. These snapshots create a transaction consistent view of a database at a particular point in time. Any changes made by other database users will not be visible within the snapshot, however they will be available when snapshot ends. Database snapshots can greatly increase parallelism, however they require the engine to retain information about database changes, and this can require a significant amount of memory. It is recommended that snapshots be used for a limited duration especially on hardware with limited resources.

Database snapshots are created using the `db.BeginSnapshot` method.

```
void BeginSnapshot () const throw (rdm_exception&);
```

This method takes no parameters and creates a static read-only view of the database.

```
void list_sensors(
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_sensor s;
    struct sensor s_fields;
    db.BeginSnapshot();
    try
    {
        s = db.Get_sensor_records();
        for(s.First(); s != Cursor::GetAfterLast(); s++)
        {
            s.Get_name(s_fields.name);
            cout << s_fields.name << endl;
        }
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Transaction Commits

The `db.End` method will end an active transaction or snapshot. Calling `db.End` will

- Commit any modifications made in a write transaction
- Free locks obtained in a Write transaction
- Free locks obtained in a Read transaction
- Release a database snapshot

```
void End (void) const throw (rdm_exception&);
Cursor_sensor add_sensor(
    Db_measurements db,
    struct sensor *sensor_fields) throw (rdm_exception&)
{
    Cursor_sensor s;
    recordType rtypes[]= {Cursor_sensor::Type()};
    db.BeginUpdate(rtypes, 1);
    try
    {
        s = db.New_sensor_recordWithFieldValues(sensor_fields);
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
}
```

```
    }
    db.End();
    return s;
}
```

Transaction Aborts

If there is an exception in the processing of a transaction block you would normally abort the transaction instead of committing it. The `db.Abort` method is similar to the `db.End` method except that it rolls back changes made in a write transaction instead of persisting them. It is illegal to call `db.Abort` for a read transaction or snapshot.

```
void Abort (void) const throw (rdm_exception&);
Cursor_sensor add_sensor(
    Db_measurements db,
    struct sensor *sensor_fields) throw (rdm_exception&)
{
    Cursor_sensor s;
    recordType rtypes[] = {Cursor_sensor::Type()};
    db.BeginUpdate(rtypes, 1);
    try
    {
        s = db.New_sensor_recordWithFieldValues(sensor_fields);
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
    return s;
}
```

Precommit

The RDM C++ interface can be involved in multi-system transactions by the use of the `db.Precommit` method. In two-phase commit protocols there is a voting phase and a commit phase. The `db.Precommit` method is used in the voting phase to validate that a transaction is able to commit. Typically if `db.Precommit` does not throw an exception then `db.End` will not throw an exception. By using `db.Precommit` your application can check if the database is able to commit a transaction before issuing the actual `db.End` or `db.Abort` command.

Consider our measurement example application, perhaps we want to periodically transfer readings from the device to an enterprise DBMS for data warehousing purposes. If the readings on the device are critical we may want to make sure that they are successfully entered in the enterprise DBMS before we delete them from the embedded database. We can do this by using a two phase-commit and having both the embedded database and the enterprise database confirm the ability to commit before actually ending the transaction.

```
void Precommit (void) const throw (rdm_exception&);
```

```
void delete_measurement(
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    bool b_do_commit = true;
    db.BeginUpdate();
    try
    {
        m.Disconnect_sensor_measurement();
        m.Delete();
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }

    try
    {
        db.Precommit();
    }
    catch (rdm_exception e)
    {
        /* Send precommit failed message to transaction broker */
        db.Abort();
        throw e;
    }

    /* Send precommit succeeded message to transaction broker */

    /* Ask transaction broker if we should End or Abort */
    if(b_do_commit == true)
        db.End();
    else
        db.Abort();
}
```

New Methods

The Db interface provides a set of generated methods for adding new records to the database. Each record declared in the database schema will have two "New" methods - `New_recordname_record` and `New_recordname_recordWithFieldValues`. The first method creates a record with field values initialized to default values. The second method creates a record with values initialized with data that is provided to the method. These methods return a singleton cursor object for the newly created record and must be called within an Update transaction block. If a record has a unique key definition it is not possible to create more than one rec-

C++ Interface Users Guide

ord with default values unless the unique key field of the created record is set to something else before the next record is created.

To create a new record with default values

```
Cursor_recname New_recname_record (void) const throw (rdm_exception&);
```

The caller of the function can use the cursor that is returned to set field values in the new record.

```
Cursor_sensor add_sensor2(
    struct sensor *sensor_fields,
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_sensor s;
    recordType rtypes[] = {Cursor_sensor::Type(), Cursor::NoMoreTypes()};
    db.BeginUpdate(rtypes);
    try
    {
        s = db.New_sensor_record();
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
    return s;
}
```

To create a new record with specified values

```
Cursor_recname New_recname_recordWithFieldValues (
    struct recname *fields) const throw (rdm_exception&);
```

```
Cursor_sensor add_sensor(
    struct sensor *sensor_fields,
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_sensor s;
    recordType rtypes[] = {Cursor_sensor::Type()};
    db.BeginUpdate(rtypes, 1);
    try
    {
        s = db.New_sensor_recordWithFieldValues(sensor_fields);
    }
    catch (rdm_exception e)
    {
        db.Abort();
    }
}
```

```
        throw e;
    }
    db.End();
    return s;
}
```

Delete Methods

The `Cursor` interface contains two methods to remove records from a database. The `Cursor.Delete` method is for deleting a record that is not actively connected to any sets. The `Cursor.DisconnectAndDelete` method will disconnect the record from any sets before it is deleted. Both methods require an active transaction context with locks on the cursor record, the `Cursor.DisconnectAndDelete` method also requires locks on any records related to the cursor record through a set.

```
Cursor Delete (void) const throw (rdm_exception&);
```

```
Cursor DisconnectAndDelete (void) const throw (rdm_exception&);
```

The first example shows how to use `Cursor.Delete` to remove a `Sensor` record that does not have any measurements related to it.

```
/* Note: this only works if the sensor has no associated measurements.  If
 * the sensor has measurements then an exception will be thrown by the
 * s.Delete call
 */
void delete_sensor(
    Cursor_sensor s,
    Db_measurements db) throw (rdm_exception&)
{
    recordType rtypes[] = {s.Type()};
    db.BeginUpdate(rtypes, 1);
    try
    {
        s.Delete();
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
}
```

The second example demonstrates how to use `Cursor.DisconnectAndDelete` to remove a `Measurement` record that is associated with a `Sensor`.

```
void delete_measurement2(
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    recordType rtypes[]= {Cursor_sensor::Type(), Cursor_measurement::Type()};
    db.BeginUpdate(rtypes, 2);
    try
    {
        m.DisconnectAndDelete();
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
}
```

Get Methods

The `Cursor` interface has a set of generated methods for retrieving non-blob field data stored in record. There is a method for retrieving all of fields in a record (`Cursor.GetFieldData`) and methods to retrieve each field value individually (`Cursor.Get_fieldname`). These need to be run within a transaction block - Update, Read, or Snapshot.

```
void GetFieldValues (struct record_name *fields) const throw (rdm_exception&);
```

```
void Get_fieldname (<type> val) const throw (rdm_exception&);
```

This example shows how to retrieve all the field values from a `Cursor_measurement` object and just the name field from a `Cursor_sensor` object.

```
void read_measurement(
    struct measurement *measurement_fields,
    struct sensor *sensor_fields,
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_sensor s;
    db.BeginRead();
    try
    {
```

```
        m.GetFieldValues(measurement_fields);
        m.Get_sensor_measurement_owner().Get_name(sensor_fields->name);
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Set Methods

Updating data in the database is done through the Cursor interface's generated Set methods methods. There is a method for setting all the fields in a record (`Cursor.SetFieldData`) and methods to set each field value individually (`Cursor.Set_fieldname`). These need to be run within an Update transaction block.

```
void SetFieldValues (const struct recordname *fields) const throw (rdm_exception&);
```

```
void Set_fieldname (<type> val) const throw (rdm_exception&);
```

This example shows how to set all the field values from a `Cursor_sensor` object.

```
void update_sensor(
    struct sensor *sensor_fields,
    Cursor_sensor s,
    Db_measurements db) throw (rdm_exception&)
{
    recordType rtypes[] = {Cursor_sensor::Type()};
    db.BeginUpdate(rtypes, 1);
    try
    {
        s.SetFieldValues(sensor_fields);
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
}
```

This example illustrates how to update the value field of a measurement record

```
void update_measurement_value (
    int32_t          value,
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    recordType rtypes[]= {Cursor_measurement::Type()};
    db.BeginUpdate(rtypes, 1);
    try
    {
        m.Set_value(value);
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
}
```

Blob Methods

Blob fields are handled slightly differently from standard fields, in addition to the Cursor generated Get and Set methods there are methods generated for retrieving and setting the size of the blob field. If you set the blob size to a value less than the current size the field will be truncated. If you set the size of a blob fields to a value larger than the existing size then the field will be padded with NULL bytes. As with standard fields Set Methods require and Update transaction and Get Methods require a Read transaction or Snapshot.

The `Cursor.Set` method is generated for each blob field to add or update data to a blob field. This method has more parameters than the `Cursor.Set` methods for standard fields. If you specify an offset in the middle of the existing data then the new data will overwrite any existing data.

```
void Set_blobfieldname (uint32_t offset, const void *buf, uint32_t size)
    const throw (rdm_exception&);
```

offset	The offset into the blob field to write the data
buf	The data to write to the blob
size	The size of the data to be written to the blob field

```
void add_measurement_raw_data (
    const void      *raw_data,
    uint32_t        size,
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    db.BeginUpdate();
    try
```

```
{
    m.Set_raw_data(0, raw_data, size);
}
catch (rdm_exception e)
{
    db.Abort();
    throw e;
}
db.End();
}
```

There is a `Cursor.Get` method generated for each blob field to read data from a blob field. This method has more parameters than the `Get` methods for standard fields

```
void Get_blobfieldname (uint32_t offset, void *buf, uint32_t len, uint32_t *size=
NULL)
    const throw (rdm_exception&);
```

offset	The offset into the blob field to read data from
buf	A buffer to write the data read
len	The amount of data to read into buf
size	The amount of data that was read into buf

```
void read_measurement_raw_data(
    void *raw_data,
    uint32_t size,
    uint32_t *bytes_read,
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    db.BeginRead();
    try
    {
        m.Get_raw_data(0, raw_data, size, bytes_read);
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

There is a `Cursor.Set_size` method generated for each blob field to change the size of the field. This can result in the field being truncated or padded with NULL bytes.

```
void Set_blobfieldname_size (uint32_t size) const throw (rdm_exception&);
```

size	The new size for the blob field
------	---------------------------------

```
void set_measurement_raw_data_size(
    uint32_t      size,
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    db.BeginUpdate();
    try
    {
        m.Set_raw_data_size(size);
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
}
```

There is a `Cursor.Get_size` method generated for each blob field to get the current size of the field.

```
void Get_blobfieldname_size (uint32_t *size) const throw (rdm_exception&);
```

`size` A variable to get the **current** size of the blob field

```
void get_measurement_raw_data_size(
    uint32_t      *size,
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    db.BeginRead();
    try
    {
        m.Get_raw_data_size(size);
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Network Model Set Methods

The Cursor interface has generated methods to support connecting and disconnecting records to sets. Methods that deal with connecting or disconnecting records to sets require Update transactions and locks on both the owner and member record types. See the RDM User's Guide for more information on Network Model Sets.

The `Cursor.Connect_setname` method connects a member to an owner's set chain.

```
void Connect_setname (const Cursor_ownertype &owner) const throw (rdm_exception&);
```

owner

A cursor positioned to the owner record the member will be connected to

```
Cursor_measurement add_measurement(
    struct measurement *measurement_fields,
    Cursor_sensor      s,
    Db_measurements    db) throw (rdm_exception&)
{
    Cursor_measurement m;
    db.BeginUpdate();
    try
    {
        m = db.New_measurement_recordWithFieldValues(measurement_fields);
        m.Connect_sensor_measurement(s);
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
    return m;
}
```

The `Cursor.Disconnect_setname` method will remove a record from a set chain.

```
void Disconnect_sensor_measurement (void) const throw (rdm_exception&);
```

```
void delete_sensor2(
    Cursor_sensor      s,
    Db_measurements    db) throw (rdm_exception&)
{
    recordType rtypes[]= {s.Type(), Cursor_measurement::Type()};
    Cursor_measurement m;
    db.BeginUpdate(rtypes, 2);
    try
    {
        m = s.Get_sensor_measurement_members();
        while(m != Cursor::GetAfterLast())
        {
            m.Disconnect_sensor_measurement();
            m++;
        }
        s.Delete();
    }
}
```

```
catch (rdm_exception e)
{
    db.Abort();
    throw e;
}
db.End();
}
```

The `Cursor.Reconnect_setname` method will remove a record from a set chain and add it to another set chain. This method is required when navigating set members via a `SetCursor` as the member record will drop out of the cursor when it is disconnected.

```
void Reconnect_sensor_measurement (const Cursor_sensor &owner) const throw (rdm_
exception&);
```

owner

A cursor positioned to the new owner record the member will be connected to

```
void move_measurement(
    Cursor_measurement m,
    Cursor_sensor      s,
    Db_measurements    db) throw (rdm_exception&)
{
    recordType rtypes[]= {s.Type(), Cursor_measurement::Type()};
    db.BeginUpdate(rtypes, 2);
    try
    {
        m.Reconnect_sensor_measurement(s);
    }
    catch (rdm_exception e)
    {
        db.Abort();
        throw e;
    }
    db.End();
}
```

The `Cursor.Get_setname_count` method will return the number of members contained in a `SetCursor` collection. The function does not require a read transaction, but can potentially read old data without one.

```
int32_t Get_sensor_measurement_count (void) const throw (rdm_exception&);
int32_t get_number_of_measurements(
    Cursor_sensor      s,
    Db_measurements    db) throw (rdm_exception&)
{
    int32_t member_count;
    db.BeginRead();
    try
    {
```

```
        member_count = s.Get_sensor_measurement_count();
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
    return member_count;
}
```

Cursor Creation Methods

In the C++ interface record searching and navigation is done through Cursor interfaces. There are four types of Cursors supported by RDM- table scan cursors, key scan cursors, and set member cursors, and singleton cursors. These cursors have a common interface and therefore can be interchanged. This section covers the generated methods that give you cursors containing a single record or groups of records.

Table Scan Cursors

A table scan cursor provides the ability to navigate all the records of a particular record type in sequential order. The exact ordering of those records will not be known to the application. In our example application this would be useful for a function that wants to find the average measurement value stored in the measurement record. The order in which the records are returned are not important in this calculation, but we do need to access every record instance.

A Table Scan cursor is returned from the Db interface with the `Db.Get_recname_records` method. There is no need for a transaction when calling `Db.Get_recname_records`, however a Read transaction or Snapshot is required to read data from a Cursor.

```
Cursor_rectype Get_recname_records (void) const throw (rdm_exception&);
```

Returns: A table scan cursor contain a collection of all the *recname* records positioned at the first record instance.

```
void get_average_measurement(
    double          *avg,
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_measurement m;
    double total;
    int32_t cur_value;
    int64_t count;

    db.BeginSnapshot();
```

```
try
{
    for(m = db.Get_measurement_records(), count=0, total=0; m != Cur-
sor::GetAfterLast(); m++, count++)
    {
        m.Get_value(&cur_value);
        total += cur_value;
    }
}
catch (rdm_exception e)
{
    db.End();
    throw e;
}
db.End();
*avg = total / count;
}
```

The `Cursor` interface has a common method `Cursor.GetRecords` to obtain a table scan cursor for its own record type. This method is similar to the generated method in the `Db` interface except that the resulting `Cursor` is not positioned at the first record instance, but rather at the location of the record that created the cursor - if that location is meaningful. Since table scan cursors are not ordered in a deterministic manner this method is of limited use.

```
Cursor GetRecords (void) const throw (rdm_exception&);
```

One potential use case would be to determine if a record was placed at the end of a file. Since RDM recycles slots when records are deleted it is possible that a record could be placed in the middle of the file or at the end. The following example uses the `Cursor.GetRecords` method to determine if a newly inserted record was placed at the end of a file. It is also an example of how you can chain method calls together.

```
Cursor_measurement add_measurement2(
    bool *b_rec_at_end,
    struct measurement *measurement_fields,
    Cursor_sensor s,
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_measurement m;

    db.BeginUpdate();
    try
    {
        m = db.New_measurement_recordWithFieldValues(measurement_fields);
        m.Connect_sensor_measurement(s);
        *b_rec_at_end = m.GetRecords().Next().AfterLast();
    }
    catch (rdm_exception e)
    {

```

```
        db.Abort();
        throw e;
    }
    db.End();
    return m;
}
```

Key Scan Cursors

Key Scan Cursors are similar to Table Scan cursors, except they are based on an b-tree index. A Key Scan Cursor allows an application to iterate through all of the records in a particular record type in index order. In addition Key Scan cursors add methods for direct record searches.

A *recname* Key Scan cursor is returned from the Db interface with the generated `Db.Get_recname_recordsBy_indexname` method. `Db.Get_recname_recordsBy_indexname` requires a Read Transaction or Snapshot, however the C++ API will implicitly create and release a Read Transaction if one is not active. RDM best practices suggest explicitly creating transactions instead of solely relying on implicit transactions. The *recname* Key Scan cursor returned by `Db.Get_recname_recordsBy_indexname` will be positioned at the first record in the cursor collection.

```
Cursor_recname Get_recname_recordsBy_indexname (void) const throw (rdm_exception&);
```

The following example shows how to display a list of all the sensors in the database ordered by the sensor name.

```
void list_sensors_by_name(
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_sensor s;
    struct sensor s_fields;
    db.BeginSnapshot();
    try
    {
        s = db.Get_sensor_recordsBy_name();
        for(s.First(); s != Cursor::GetAfterLast(); s++)
        {
            s.Get_name(s_fields.name);
            cout << s_fields.name << endl;
        }
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

C++ Interface Users Guide

It is also possible to obtain a Key Scan Cursor directly from another cursor by using the generated method `Cursor.GetRecordsBy_indexname`. This result is similar to the method in the Db interface except that the resulting Cursor is not positioned at the first record instance, but at the location of the record from which we derived the new cursor from.

```
Cursor_recname GetRecordsBy_indexname() const throw (rdm_exception&);
```

The following example uses the `GetRecordsBy_indexname` from a Cursor to display all sensor readings that have occurred after the current reading in the cursor (based on the `k_reading` index).

```
void list_subsequent_measurements(
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_measurement m_later;
    struct measurement m_fields;
    db.BeginSnapshot();
    try
    {
        m_later = m.GetRecordsBy_k_reading();
        for(m_later.Next(); m_later != Cursor::GetAfterLast(); m_later++)
        {
            m_later.GetFieldValues(&m_fields);
            display_measurement(&m_fields);
        }
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

The Cursor interface also provides generated methods for performing searches on indexed fields. For each index defined in the database there will be a `Cursor.KeyFindBy_indexname` method generated. This method will reposition the cursor to the first occurrence of the key value in the cursor collection. This method repositions the cursor to the first occurrence of the key value in the cursor collection, but it does not change the Cursor type. If you call `KeyFindBy_indexname` from a Record Scan cursor any subsequent Next or Previous calls will be based on the Cursor order and not on the index order. `Cursor.KeyFindBy_indexname` requires a Read Transaction.

When calling `KeyFindBy_indexname` from a Key Scan cursor, if you do not get an exact match the Cursor will not be at a valid record, but it will be positioned at the index location where a match would have been located. Consider an integer index containing the values 1, 2, 7, and 8, a `KeyFindBy_indexname` with a value of 5 would result in a positioning the index to a point where a call to Previous would position the Cursor to the record with the '2' index value and a call to Next would position the Cursor to the record with the '7' index value. Attempted to read/write/delete from a Cursor that is not currently positioned at a record will result in a `rdm_not_at_record_exception` being thrown.

C++ Interface Users Guide

When working with a Set Member Cursor the `KeyFindBy_indexname` will only find records that are in the cursor. While Key Scan and Record Scan cursors contain all of the records in a table, Set Member cursors only contain member records associated with a particular owner.

Since a Singleton cursor by definition contains only a single record, the `KeyFindBy_indexname` method for a Singleton Cursor is not meaningful.

```
void KeyFindBy_indexname (field_type *field_value) const throw (rdm_exception&);
```

This method has one parameter

field_value The index value that will be used to position the cursor

This example shows how to position a Sensor cursor to a particular sensor record based on the indexed name field. For this function it does not matter if the cursor is a Record Scan or Key Scan cursor.

```
void lookup_sensor(
    char          *name,
    Cursor_sensor s,
    Db_measurements db) throw (rdm_exception&)
{
    recordType rtypes[] = {s.Type()};
    db.BeginRead(rtypes, 1);
    try
    {
        s.KeyFindBy_name(name);
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Compound keys have an addition overloaded `Cursor.KeyFindBy_indexname` method

```
void KeyFindBy_indexname (int32_t nFields, uint32_t partialStrLen,
    field_type *field_value) const throw (rdm_exception&);
```

nFields The number of fields to use in the search. If this parameter is zero, the function searches all fields. If this parameter is a non-zero value, the function uses only the number of fields indicated.

partialStrLen The partial string length in bytes. If this parameter is a non-zero value, the function uses only this many bytes of the last field being considered (depending on *nFields* above).

field_value The index value that will be used to position the cursor

This example shows how to position a Measurement cursor to a particular record based on the `k_reading` compound key. For this function it does not matter if the cursor is a Record Scan or Key Scan cursor.

```
void lookup_measurement(
    struct measurement_k_reading *reading,
    Cursor_measurement          m,
    Db_measurements            db) throw (rdm_exception&)
{
    recordType rtypes[]= {m.Type()};
    db.BeginRead(rtypes, 1);
    try
    {
        m.KeyFindBy_k_reading(2, 0, reading);
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Set Member Cursors

Set Member Cursors differ from Table Scan Cursors and Key Scan Cursors as they can only be obtained from another Cursor and they typically contain a subset of a table's total records. A Set Member Cursor allows an application to navigate network model set relationships. The Set Member Cursor contains a collection of all the member records that are associated with a particular owner record. There are two generated methods to obtain Set Member Cursors.

Every record that is an "owner" in a set relationship will have a generated method called `Cursor.Get_setname_members`. This method will return a Set Member collection containing all of the member records associated with the current record in the owner's cursor collection. The cursor will be positioned at the first record in the collection.

```
Cursor_rectype Get_setname_members (void) const throw (rdm_exception&);
```

This example displays all of the measurements associated with a sensor through the `sensor_measurement` set.

```
void list_sensor_measurements(
    Cursor_sensor    s,
    Db_measurements db) throw (rdm_exception&)
{
    int32_t          ii;
    Cursor_measurement m;
    struct measurement m_fields;
    db.BeginRead();
    try
    {
```

```

m = s.Get_sensor_measurement_members();
for(m.First(), ii=0; m != Cursor::GetAfterLast(); m++, ii++)
{
    m.GetFieldValues(&m_fields);
    cout << "Reading: " << ii << endl;
    cout << "Time: " << m_fields.time << endl;
    cout << "Value: " << m_fields.value << endl;
}
}
catch (rdm_exception e)
{
    db.End();
    throw e;
}
db.End();
}

```

Every record that is a "member" in a set relationship will have a generated method called `Cursor.Get_setname_siblings`. This method will return a Set Member collection containing all of the member records with the same owner as the original cursor. The new cursor collection will be positioned at the location of the cursor from which we derived the new Set Member cursor and not the first member.

```

Cursor_rectype Get_setname_siblings (void) const throw (rdm_exception&);

```

In our test application sensors store their measurement readings using a set. This set will put all new readings at the end of the set chain. By using the `Get_sensor_measurement_siblings` method we can find all of sensor's readings that came before or after a particular reading without needing an index.

```

void list_prior_measurements_for_sensor(
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    int32_t ii;
    Cursor_measurement m_prior;
    struct measurement m_fields;
    db.BeginRead();
    try
    {
        m_prior = m.Get_sensor_measurement_siblings();
        for(m_prior.Prev(), ii=0; m_prior != Cursor::GetBeforeFirst(); m_prior--,
ii++)
        {
            m_prior.GetFieldValues(&m_fields);
            cout << "Reading: " << ii << endl;
            cout << "Time: " << m_fields.time << endl;
            cout << "Value: " << m_fields.value << endl;
        }
    }
    catch (rdm_exception e)

```

```
{
    db.End();
    throw e;
}
db.End();
}
```

Singleton Cursors

The final type of cursor is a singleton cursor which can only contain one record. For convenience a singleton can be used just as any other type of cursor, however it will never yield more than one record.

When creating a new record the generated `Db.New_recordname_record` and `Db.New_recordname_recordWithFieldValues` methods return a **Singleton Cursor** containing the newly created record.

Each record that is defined as a member in a set relationship will have a method generated to obtain a singleton cursor containing the records "Owner".

```
Cursor_ownerrec Get_setname_owner (void) const throw (rdm_exception&);
```

Returns: A singleton cursor containing the owner record.

This example shows how to get a singleton cursor containing the related sensor record for any measurement stored in our sample database.

```
Cursor_sensor get_sensor_for_measurement(
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_sensor s;
    db.BeginRead();
    try
    {
        s = m.Get_sensor_measurement_owner();
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
    return s;
}
```

Each cursor also contains a `Cursor.GetThis` method that will return a **Singleton Cursor** for the current record in the cursor collection. This allows an application to save a reference to a particular record for later use.

```
Cursor GetThis (void) const throw (rdm_exception&);
```

Returns: A singleton cursor containing the owner record.

Special Singletons

There are two special singletons 'BeforeFirst' and 'AfterLast' used for cursor navigation. These special singletons can be obtained using static methods in the Cursor interface and identify when a Cursor is positioned before the first record in the collection or after the final record in the collection.

```
static Cursor GetBeforeFirst (void) throw ();
```

Returns: The BeforeFirst special singleton

```
static Cursor GetAfterLast (void) throw ();
```

Returns: The AfterLast special singleton

This example demonstrates how to navigate through a cursor collection and uses the GetAfterLast singleton to identify when we have processed the final record in a cursor collection.

```
void list_sensors(
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_sensor s;
    struct_sensor s_fields;
    db.BeginSnapshot();
    try
    {
        s = db.Get_sensor_records();
        for(s.First(); s != Cursor::GetAfterLast(); s++)
        {
            s.Get_name(s_fields.name);
            cout << s_fields.name << endl;
        }
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Cursor Navigation Methods

A Cursor obtained from a Db interface is always positioned to the first record in the sequence. A Cursor obtained from another Cursor is typically positioned to the same record as the originating Cursor. Once a Cursor had been obtained there are methods to navigate through the collections of records contained by the Cursor. All Cursors in the RDM C++ Interface are considered live Cursors. Any changes made to the database will affect all active Cursors. If you need a Cursor to remain constant you must ensure a transaction is kept for the life of the Cursor.

First

It is possible to position a Cursor to the first record in the Cursor collection with the `Cursor.First` method. This method returns a copy of the Cursor which allows it to be chained in a series of method calls.

```
Cursor First (void) const throw (rdm_exception&);
```

Returns: The Cursor you are using positioned to the first record in the collection.

The following example demonstrates the use of the `Cursor.First` method in looping through all of the records in a Cursor collection.

```
void list_sensors(
    Db_measurements db) throw (rdm_exception&)
{
    Cursor_sensor s;
    struct sensor s_fields;
    db.BeginSnapshot();
    try
    {
        s = db.Get_sensor_records();
        for(s.First(); s != Cursor::GetAfterLast(); s++)
        {
            s.Get_name(s_fields.name);
            cout << s_fields.name << endl;
        }
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Last

It is possible to position a `Cursor` to the last record in the `Cursor` collection with the `Cursor.Last` method. This method returns a copy of the `Cursor` which allows it to be chained in a series of method calls.

```
Cursor Last (void) const throw (rdm_exception&);
```

Returns: The `Cursor` you are using positioned to the last record in the collection.

The following example demonstrates the use of the `Cursor.Last` method by displaying the last 5 records in a `Set` collection

```
void list_latest_sensor_measurements (
    Cursor_sensor s,
    Db_measurements db) throw (rdm_exception&)
{
    int32_t ii;
    Cursor_measurement m;
    struct measurement m_fields;
    db.BeginRead();
    try
    {
        m = s.Get_sensor_measurement_members();
        for(m.Last(), ii=0; m != Cursor::GetBeforeFirst() && ii < 5; m--, ii++)
        {
            m.GetFieldValues(&m_fields);
            cout << "Reading: " << ii << endl;
            cout << "Time: " << m_fields.time << endl;
            cout << "Value: " << m_fields.value << endl;
        }
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Next

Position a `Cursor` to the next record in the collection by using the `Cursor.Next` method. The `'++'` operator can be used as a shortcut for the `Cursor.Next` method. This method returns a copy of the `Cursor` which allows it to be chained in a series of method calls.

```
Cursor Next (void) const throw (rdm_exception&);
```

```
Cursor operator ++ (void) const throw (rdm_exception&);
```

```
Cursor operator ++ (int) const throw (rdm_exception&);
```

Returns: The Cursor you are using positioned to the next record in the collection.

The following example demonstrates the use of chaining `Cursor.Next` method to determine if a record is the final record in a Cursor Collection

```
Cursor_measurement add_measurement2(  
    bool *b_rec_at_end,  
    struct measurement *measurement_fields,  
    Cursor_sensor s,  
    Db_measurements db) throw (rdm_exception&  
{  
    Cursor_measurement m;  
  
    db.BeginUpdate();  
    try  
    {  
        m = db.New_measurement_recordWithFieldValues(measurement_fields);  
        m.Connect_sensor_measurement(s);  
        *b_rec_at_end = m.GetRecords().Next().AfterLast();  
    }  
    catch (rdm_exception e)  
    {  
        db.Abort();  
        throw e;  
    }  
    db.End();  
    return m;  
}
```

Prev

Position a Cursor to the previous record in the collection by using the `Cursor.Prev` method. The `'--'` operator can be used as a shortcut for the `Cursor.Prev` method. This method returns a copy of the Cursor which allows it to be chained in a series of method calls.

```
Cursor Prev (void) const throw (rdm_exception&);
```

```
Cursor operator -- (void) const throw (rdm_exception&);
```

```
Cursor operator -- (int) const throw (rdm_exception&);
```

C++ Interface Users Guide

Returns: The Cursor you are using positioned to the previous record in the collection.

The following example demonstrates the use `Cursor.Prev` method and `'--'` operator to display prior measurements from a sensor.

```
void list_prior_measurements_for_sensor(
    Cursor_measurement m,
    Db_measurements db) throw (rdm_exception&)
{
    int32_t ii;
    Cursor_measurement m_prior;
    struct measurement m_fields;
    db.BeginRead();
    try
    {
        m_prior = m.Get_sensor_measurement_siblings();
        for(m_prior.Prev(), ii=0; m_prior != Cursor::GetBeforeFirst(); m_prior--,
ii++)
        {
            m_prior.GetFieldValues(&m_fields);
            cout << "Reading: " << ii << endl;
            cout << "Time: " << m_fields.time << endl;
            cout << "Value: " << m_fields.value << endl;
        }
    }
    catch (rdm_exception e)
    {
        db.End();
        throw e;
    }
    db.End();
}
```

Example 1: Cursor.Prev method

Count

Each cursor has a `GetCount` method which will returns the number of items contained in the Cursor collection. By definition the `GetCount` method for a Singleton Cursor will always return 1. The `GetCount` method for the special Cursors (BeforeFirst/AfterLast) will always return 0.

A read transaction is not required for this method, but it is possible to retrieve out of data information without one.

```
uint64_t GetCount (void) const throw (rdm_exception&);
```

Returns: The number of items contained in the Cursor collection

The following example demonstrates the use `Cursor.GetCount` method to retrieve the number of sensors records in the database.

```
int64_t get_number_of_sensors(  
    Cursor_sensor s,  
    Db_measurements db) throw (rdm_exception&)  
{  
    int64_t sensor_count;  
    db.BeginRead();  
    try  
    {  
        sensor_count = s.GetCount();  
    }  
    catch (rdm_exception e)  
    {  
        db.End();  
        throw e;  
    }  
    db.End();  
    return sensor_count;  
}
```

Cursor Comparison Methods

A cursor can be positioned at any record in the sequence, it can also be positioned before the first record or after the last record. At times we may want to compare the position of one cursor with the position of another. To support this the C++ interface provide a set of methods and overloaded operators that compare cursor positions.

Equal

The '==' operator and the `Cursor.At` method perform a comparison for equality. They will return true only if the positions that are being compared represent the same record (or both are before the first record or after the last record). The cursors involved in the comparison do not need to be of the same type.

```
bool At (const Cursor& cursor) const throw (rdm_exception&);
```

```
bool operator == (const Cursor& cursor) const throw (rdm_exception&);
```

Not Equal

The '!=' operator and the `Cursor.NotAt` method perform a comparison for inequality. They will return true only if the positions that are being compared do not represent the same record. The cursors involved in the comparison do not need to be of the same type.

```
bool NotAt (const Cursor& cursor) const throw (rdm_exception&);
```

```
bool operator != (const Cursor& cursor) const throw (rdm_exception&);
```

Less Than

The '`<`' operator and the `Cursor.Before` method perform less than comparison for two cursors. They will return true only if the position in the left operand comes before the position in the right operand (as it is positioned in the left operands cursor sequence). The cursors involved in the comparison do not need to be of the same type.

A key scan cursor (`cursor1`) and a record scan cursor (`cursor2`) may contain the exact same set of records, but the records typically occur in different orders. A comparison between these cursors will be based on the order of the left operand. This means if `cursor1 < cursor2` (key order) it does not necessarily mean `cursor2 > cursor1` (record order)

```
bool Before (const Cursor& cursor) const throw (rdm_exception&);
```

```
bool operator < (const Cursor& cursor) const throw (rdm_exception&);
```

Less Than Or Equal

The '`<=`' operator and the `Cursor.BeforeAt` method perform a less than or equal comparison for two cursors. They will return true if the '`<`' operator or the '`==`' operator return true.

```
bool BeforeAt (const Cursor& cursor) const throw (rdm_exception&);
```

```
bool operator <= (const Cursor& cursor) const throw (rdm_exception&);
```

Greater Than

The '`>`' operator and the `Cursor.After` method perform greater than comparison for two cursors. They return the negated result of the '`<=`' operator.

```
bool After (const Cursor& cursor) const throw (rdm_exception&);
```

```
bool operator > (const Cursor& cursor) const throw (rdm_exception&);
```

Greater Than Or Equal

The '`>=`' operator and the `Cursor.FasterAt` method perform a greater than or equal comparison for two cursors. They will return the negated result of the '`<`' operator.

Error Handling

Very simple applications can get away without any error handling, if an error occurs an exception will be thrown and the application will terminate. For non-trivial applications this approach is not appropriate, the application needs to check for errors and take appropriate actions when errors are detected.

In the RDM C++ interface there are two kinds of errors, expected errors and unexpected errors. Expected errors come from the use of the C++ API and an application must be coded to handle them. An example of an expected error is the inability to start a 'write transaction' because of locking contention. Only one thread is allowed to have a write transaction for a specific table, so an application that is requesting a write transaction must expect that their request may timeout and an error will occur. In this case the application may want to write some information out to a log file and then attempt to restart the transactions. Unexpected errors are more difficult to handle and may require the program to terminate. This would be the case if the application runs out of memory or if a database is corrupted due to a hardware error or software bug.

Exceptions

Error handling in the C++ interface is done through the use of exceptions. The C++ interface defines the `rdm_exception` class for all interface specific errors. Expected errors throw a class derived from `rdm_exception` while unexpected errors throw the base `rdm_exception` class. The following example demonstrates how we can use exception handling to verify the type of record a cursor collection contains. In the try block we attempt to cast a generic `Cursor` to record specific `Cursor_sensor`. The `Cursor_sensor` copy constructor will instantiate a `Cursor_sensor` with the `Cursor` as the template. The copy constructor will throw an `rdm_invalid_type_exception` if the `Cursor` is not a sensor cursor. Any other type of exception is not handled by the `is_sensor` function will need to be handled by the caller.

```
bool is_sensor(
    Cursor c)
{
    try
    {
        (Cursor_sensor) c;
    }
    catch (rdm_invalid_type_exception e)
    {
        return false;
    }
    return true;
}
```

Glossary

B

B-tree

Also called a multiway tree, a B-tree is a fast data-indexing method that organizes the index into a multi-level set of nodes. Each node contains a sorted array of key values (the indexed data). Two important properties of a B-tree are that all nodes are at least half-full and that the tree is always balanced (that is, an identical number of nodes must be read in order to locate all keys at any given level in the tree). A well-organized B-tree will have only three or four levels.

buffer

An in-memory store of data read from a disk file, in which database operations are performed.

C

cache

A set of buffers used to optimize database input and output operations. All RDM Embedded database input and output is performed using a cache.

combine

The concatenation of the members of two or more set types into one set type.

commit

The point at which database changes made during a single transaction are actually written to the database files.

compound key

A key field composed of any combination of fields (not necessarily contiguous) from a record. Each field of a compound key may be stored in ascending or descending order.

connect

The process of inserting a member record occurrence into a set occurrence.

currency tables

A table of database addresses maintained by the RDM Embedded runtime system for controlling record access and set navigation. The currency tables consist of the current member table, current owner table, and the current record.

current database

The database that is currently accessible by the RDM Embedded runtime functions when multiple databases have been opened. The current database is changed by the database number function argument or by function `d_setdb`.

current member

Contains, for each set, the database address of a record occurrence that is a valid member of that set. Usually, the current member of a set is the last record accessed using a set navigation function (`d_findfm`, `d_findlm`, `d_findnm`, or `d_findpm`).

current owner

Contains for each set, the database address of a record occurrence that is a valid owner of that set. Usually, the current owner of a set is established using the set navigation function `d_findco` or by using a currency manipulation function.

current record

Contains the database address of the most recently accessed record instance.

D

data field

A field represents the basic unit of information storage in a database and is always defined to be an element of a record. A field has associated with it attributes such as name, type (for example, char or int), and length. Other terms used for field include: attribute, entity, or column.

data file

An RDM Embedded file defined in a DDL specification that contains occurrences of one or more record types.

database

An organized collection of related files.

database address

The location in the database of a record occurrence, frequently referred to as a `DB_ADDR`. Composed of two numbers: the file index and the slot within the file. Either 4 or 8 bytes long.

database definition language

A programming-like language used to define the structure and content of a database. RDM Embedded's Database Definition Language has been designed to be used with the C programming language.

DDL

A programming-like language used to define the structure and content of a database. RDM Embedded's Database Definition Language has been designed to be used with the C programming language.

deadlock

A situation in which multiple processes accessing the same database each hold locks needed by the other processes in such a way that none of the processes can proceed. Sometimes called deadly embrace.

delete chain

A linked list containing deleted records or nodes to be reused when a new record or node is created.

derived revision

A revision that can be derived from a comparison of the source and destination database dictionary files.

destination database

The db_REVISE-created database that stores the specified revisions.

dictionary

A repository containing a definition of the content and structure of a database. It is used by the RDM Embedded runtime library functions for accessing and manipulating information from that database.

disconnect

The process of removing a member record from a set occurrence.

document root

The path to the directory under which all files will be stored. Within the domain of one TFS, no files outside of this path may be accessed.

domain name

The "name" of a computer which has visibility to another computer. This may be a published name available on DNS servers and across the Internet, or an internal network name visible only within a workgroup. The "ping" utility must be able to locate the IP address associated with this name. In RDM Embedded, a server (tfserver, dbmirror, dbrep, or dbrepSQL) may be located through the domain name of the computer it is running on, together with the port on which it is listening. A special domain name, "localhost" always refers to the same computer as the application is running on (IP address is always 127.0.0.1).

E

environment variable

A programmer-specified operating system parameter that is used to identify configuration information to the runtime system.

F

field

A field represents the basic unit of information storage in a database and is always defined to be an element of a record. A field has associated with it attributes such as name, type (for example, char or int), and length. Other terms used for field include: attribute, entity, or column.

file

The primary physical storage unit into which a database is organized. In RDM Embedded, files are used to store records and keys.

H

hierarchical database model

A data representation in which the relationships between record types are formed from parent-child structures, such that a record type may have many child relationships but only one parent relationship.

I

index

A set of key values through which rapid retrieval of a record is provided, similar to the index of a book. The term is often used synonymously with key file.

J

join

The creation of one record type from a hierarchy of record types.

K

key

A field through which rapid and/or sorted access to a record is desired.

key file

A file that only contains keys. It may, in fact, contain more than one index because multiple key types can be contained in a single RDM Embedded key file.

key scan

The process of performing an ordered traversal through all (or a subset of all) occurrences of a given key field.

L

localhost

A special Domain Name that always refers to the computer on which the application software is running. It is the default domain name used by RDM Embedded utilities and runtime library.

lock

A multi-user database synchronization mechanism, used to prevent simultaneous updates to shared data. Locks can be applied to the entire database or to files.

logging

The process of making a copy of the database changes made during a transaction prior to a commit. Logging is used to support the ability to perform a recovery in the event a failure occurs during a commit.

M

many-to-many relationship

A relationship between two record types, A and B, such that for each occurrence of type A, there are many related occurrences of type B and, for each occurrence of type B, there are many related occurrences of type A. In RDM Embedded, many-to-many relationships can be implemented using two one-to-many sets through a third, intersection record type.

member of set

Specifies a one-to-many relationship between record types. One occurrence of the owner record type is related to many occurrences of a member record type. Also called a set type.

member pointer

Stores set membership linkage information. There is one member pointer stored with a record per set for which the record is a member. Each one contains the database addresses of the owner record, previous member in the set, and next member in the set.

N

navigation

The process of retrieving records from a database by moving through various navigational methods. Methods include set navigation, key scanning, and record-type scanning.

network database model

A data representation in which the relationships are explicitly defined and maintained through sets of owner/members, where any given record type may be the owner of multiple types of sets and the member of multiple types of sets. Multiple set membership distinguishes the Network database model from the Hierarchical database model.

node

A component of a B-tree, consisting of a page of sorted keys stored in a key file.

normalize

The elimination of redundant record instances that own a new set, resulting in a one-to-many relationship.

O

occurrence

One record instance within a record type, specifically associated with record type scanning (`d_recrfst`, `d_recnxt`, `d_recpv`, `d_reclast`), where the current occurrence of a record type is used to bookmark the position on a record type scan. Record occurrences are ordered by their physical appearance in a data file. The current occurrence is not the same as the current record, although the current record will also be set by the scanning functions.

owner of set

Specifies a one-to-many relationship between record types. One occurrence of the owner record type is related to many occurrences of a member record type. Also called a set type.

P

page

Files are blocked into contiguous fixed-length segments called pages. A page is the unit of database I/O performed in RDM Embedded.

path name

The sequence of directories in a hierarchical file system that must be traversed to locate a particular file.

pointer

In a database, a pointer is data stored in a record occurrence that provides the necessary information for locating related record occurrences. In a C program, a pointer is a variable that contains a memory address.

port

Together with an IP address, a port number uniquely identifies an endpoint by which a TCP/IP connection can be made to another program. In RDM Embedded, each server (tfserver, dbmirror, dbrep or dbrepsql) identifies the port number that should be used to locate it. The IP address is normally obtained through a domain name lookup (e.g. tfs.raima.com is a domain name, and its IP address is 198.168.140.200).

process

An independently executing task or program. An individual execution of an RDM Embedded application program.

projection

The placement of fields from one record type into one or more new record types.

Q

queue

A first-in-first-out waiting list. Lock requests for a locked resource will be placed at the end of a queue. When the locked resource becomes available, the first lock request on the queue will be granted.

R

record

Used synonymously with record type or record occurrence depending on the context in which the term is used.

record occurrence

One individual instance in a database of a record of a particular type. A database consists of many occurrences of many different record types. For example, an employee record type may consist of the fields name, employee_id, job_title, and pay. An employee record occurrence could be "name: Jones, Jim; employee_id: c87101, job_title: engr, pay: 3400".

recovery

The process of completing the transaction of a process that failed during a commit.

redundant data

Identical data that is stored in multiple locations in a database. Typically used to form relationships between tables in a relational database management system.

relational database model

A data representation in which a database is viewed as consisting of two-dimensional tables, each composed of one or more columns. Inter-table relationships are defined through use of common column names and data. Tables and columns are analogous to RDM Embedded records and fields, respectively.

remote procedure call

A programming mechanism that makes a library call appear to operate in the program space of an application, even though the actual function exists in the program space of another program (called a "server"). A client application places a function identifier and parameter contents into a packet that is first transferred to the server, with results (return code, return parameter values) transferred back to the caller.

Revision Definition Language

The RDL supplies information to db_REVISE that cannot be derived from a comparison of the source and destination dictionary files.

root node

The top or start node of a B-tree.

RPC

A programming mechanism that makes a library call appear to operate in the program space of an application, even though the actual function exists in the program space of another program (called a "server"). A client application places a function identifier and parameter contents into a packet that is first transferred to the server, with results (return code, return parameter values) transferred back to the caller.

runtime system

The RDM Embedded C language library functions that perform all of the database access required by an application program while it is executing.

S

schema

A conceptual model of the structure of a database that defines the data contents and relationships. A database definition language specification is an implementation of a particular schema.

set

Specifies a one-to-many relationship between record types. One occurrence of the owner record type is related to many occurrences of a member record type. Also called a set type.

set occurrence

An individual instance of a set in which one owner record occurrence has one or more member record occurrences connected to it.

set pointer

Stores set ownership linkage information. There is one set pointer stored with a record per set for which the record is an owner. Each one contains a count of the number of members in the set, the database address of the first member record occurrence, and the database address of the last member record occurrence in the set.

set scan

The process of performing an ordered traversal through all (or a subset of all) member record occurrences of a given set occurrence.

slot

A position in a data or key file for storage of a single record or key occurrence.

source database

The database containing the data that is to be revised. This database is used in a read-only manner.

specified revision

A revision requiring specification by an RDL statement.

split

The separation of a multiple-member set type into two or more set types.

static revision

A revision that can be performed without changing the existing database content or structure.

synchronization

The process of ensuring that, in a multi-user database environment, updates to shared data are performed serially, one user at a time.

system record

A special record type used to define the "top" record in a network database. There is only one occurrence of the system record in a database. It is defined by naming "system" as a set owner in one or more set definitions in the DDL. When a database is opened, the system

record, if it exists, is set as the current owner of all sets for which it is named as owner. It may not be a set member.

T

task

In an RDM Embedded Application, a task is a block of allocated memory that stores the complete database context for a thread of execution. It must be allocated through the `d_open-task` function and closed through the `d_closetask` function. A task represents one user in a multi-user environment. A task can also represent one database transaction, with all locks and database updates associated with the transaction.

TFS

A software component within the RDM Embedded system that maintains safe multi-user transactional updates to a set of files, and responds to page requests. The `tfserver` utility links to the TFS to allow it to run as a separate utility. The TFS may also be linked directly into an application in order to avoid the RPC overhead of calling a separate server.

thread

An independent flow of control within a computer operating system. Differentiated from a Process in that a process may contain one or more threads. Threads within the same process share common (or global) data but have their own stacks, which keeps track of the thread's context. In RDM Embedded Applications, each thread must be associated with its own task variable, and is treated as a separate user in a multi-user environment.

timeout

An event that occurs when a lock request has waited on a queue longer than a pre-determined amount of time. It is used to avoid deadlock.

transaction

A group of related database changes that are written to the database as a single unit during a commit. The logical consistency of a database is maintained by placing all related updates within transactions.

transactional file server

A software component within the RDM Embedded system that maintains safe multi-user transactional updates to a set of files, and responds to page requests. The `tfserver` utility links to the TFS to allow it to run as a separate utility. The TFS may also be linked directly into an application in order to avoid the RPC overhead of calling a separate server.

W

working database

A temporary database created by db_REVISION for use only during the database revision process. db_REVISION removes the working database when the revision process is complete.

Index
