RDM Embedded 10.1

RDMe DataFlow and Mirroring User Guide

Trademarks

Contents

Contents ... 3
Introduction ... 5
DataFlow Architecture ... 6
Mirroring Process ... 6
Replication Process ... 9
Database Storage Location ... 11
Synchronous Mirroring ... 13
DataFlow Utilities ... 14
4.1 Dbmirror Usage ... 15
4.2 Dbrep, dbrepsql Usage ... 16
4.3 Installation as Service or Daemon Process ... 16
4.3 Dbget Usage ... 17
4.4 Schemaxlate Usage ... 18
Mirroring Setup ... 19
Replication Setup ... 23
Opening Slave Databases from Programs ... 25
Advanced Topics ... 26
8.1 Differences Between Master and Slave ... 26
8.2 In-Memory to On-Disk ... 26
8.3 Replication-Only Changes for RDM Embedded Databases ... 26
8.4 Replication-Only Changes for SQL Databases ... 28
8.5 Slave Database Setup ... 29
8.6 Synchronization Issues ... 33
DataFlow and Mirroring API Functions ... 35
d_dbmir_init ... 36
d_dbrep_connect ... 39
d_dbrep_disconnect ... 42

d_dbrep_init 44
d_dbrep_start 47
d_dbrep_stop 49
d_dbrep_term 51
d_dbrepsql_init 53
DataFlow and Mirroring Utilities 56
dbmirror 57
dbrep, dbrepsql 59
dbget 61
schemaxlate 63

Introduction

RDM Embedded supports two unique but related methods for maintaining nearly real-time copies of databases in additional locations, referred to as *mirroring* and *replication*.

If you have licensed a **HA** package, then you have mirroring functionality. If you have licensed a **Data Flow** package, then you have both mirroring and replication. A basic **Engine** package, or the **Distributed** package, does not have either mirroring or replication. See Packages in RDM Embedded.

Both mirroring and replication use the same terminology for the roles of databases: the original, updateable database is called the *master*. From one master database, one or more *slave* copies can be created and dynamically maintained. The terminology comes from the idea that the master database controls the generation of data, and the slaves respond only when changes have been made on the master.

Mirroring

To mirror a database is to create a byte-for-byte copy of a database at a different location. Mirroring is different than copying or backing up a database in that a mirror database is updated at the same time as the original database (synchronous) or as quickly as possible after the original (asynchronous) database is updated. Page images from the master are applied to the slave(s) to implement mirroring.

There are three main purposes for mirroring:

1. To maintain another copy of a database for safe-keeping. The backup copy may be an on disk copy of an in-memory master database.
2. To offload reading of a database to another computer.
3. To be prepared to switch processing to another computer if the primary computer fails. This is often referred to as a Highly-Available (HA) database.

Replication

To replicate a database is to perform the same actions on another database as were originally applied to the master database.

The primary purpose of replication in the RDM Embedded environment is to send updates from the master database to SQL databases. Initially, RDM Embedded replication can be applied to RDM Server, MySQL, Oracle, and MS SQL Server.

Another purpose of replication is to allow additional keys or tables to exist in the slave database that don't exist, for performance reasons, in the master database. Also, circular tables may be used for efficiency in the master database, and turned into permanent logs on the slave.

A utility can be used to generate SQL DDL as needed to define a SQL database that can receive replicated data. Once the SQL database exists, the replication capability of RDM Embedded can generate SQL INSERT and DELETE statements that make sure that the SQL database contains the same data as the original RDM Embedded database.

DataFlow Architecture

The RDM Runtime always generates change log files as part of the process of committing a transaction. These log files can also be used, after being committed to the original database, as the basis for updating mirrors of the original database. Whenever a log is written to the original, the log is copied to locations where mirrors exist, and then are applied to the mirror databases. This re-use of log files means that the databases are byte-for-byte identical, and therefore must be used on computer architectures that have the same integer byte ordering. The use of page images in the log files means that transaction integrity is maintained and the recovery mechanism is very reliable.

If replication is needed, a second artifact is required, an action log file. The action log file is generated by the runtime library when the configuration (shown below) indicates that the database is a replication master. The action log files, herein called replication log files, are transmitted to the locations of replication slaves through the same mechanism used for the change log files. A replication log differs from a change log in that it is a journal of actions (e.g. create record, connect record to set, delete record), where a change log is a set of page images as they exist after all of the actions have been completed at the end of a transaction.

SQL updates can be generated from replication logs, but not from change logs.

The mirroring process is primarily *asynchronous*, meaning that transactions applied to the master database will be propagated to the slaves as soon as possible. A *synchronous* mirroring process forces an acknowledgement from the slave that the transaction has been successfully applied prior to allowing the master to be updated again. Synchronous mirroring may be required in situations where the master and slave *must* be kept identical at all times, but it comes with a performance price. A restriction on synchronous mirroring is that a master may only have one synchronous slave, even if multiple slaves exist - all additional slaves are asynchronous.

Replication is always asynchronous.

Asynchronous mirroring and replication are designed to optimize the process of maintaining extra copies of a database, allowing the master to be updated at a maximum rate without being impeded by the performance of the slaves. There can be "bursts" of changes to the master, and time afterward for the slave(s) to catch up. It also provides for intermittent connections, meaning that a slave database may be "offline" for a period of time and can get caught up (sometimes called a "synchronize" operation) when it reconnects. Log files are kept with the master database for the purpose of allowing intermittent connections to catch up. The time a log file is kept with the master can be configured such that it is deleted after a certain age. Logs can also be deleted, oldest first, if the total space taken by them exceeds a configurable size.

A slave may be on the same computer as the master (simply a different location), or on a different computer. The network connection can be a fast LAN within an office, or Internet from anywhere in the world (requires connectivity).

A utility named `dbmirror` is used to move log files from the master to the slave database. A slave database configuration will also run `dbmirror` if it is a mirrored slave, or `dbrep` if it is a replicated RDM Embedded slave, or `dbrepsql` if it is a replicated SQL slave.

Mirroring Process

Figure 12-1 below shows the generalized view of the mirroring process.

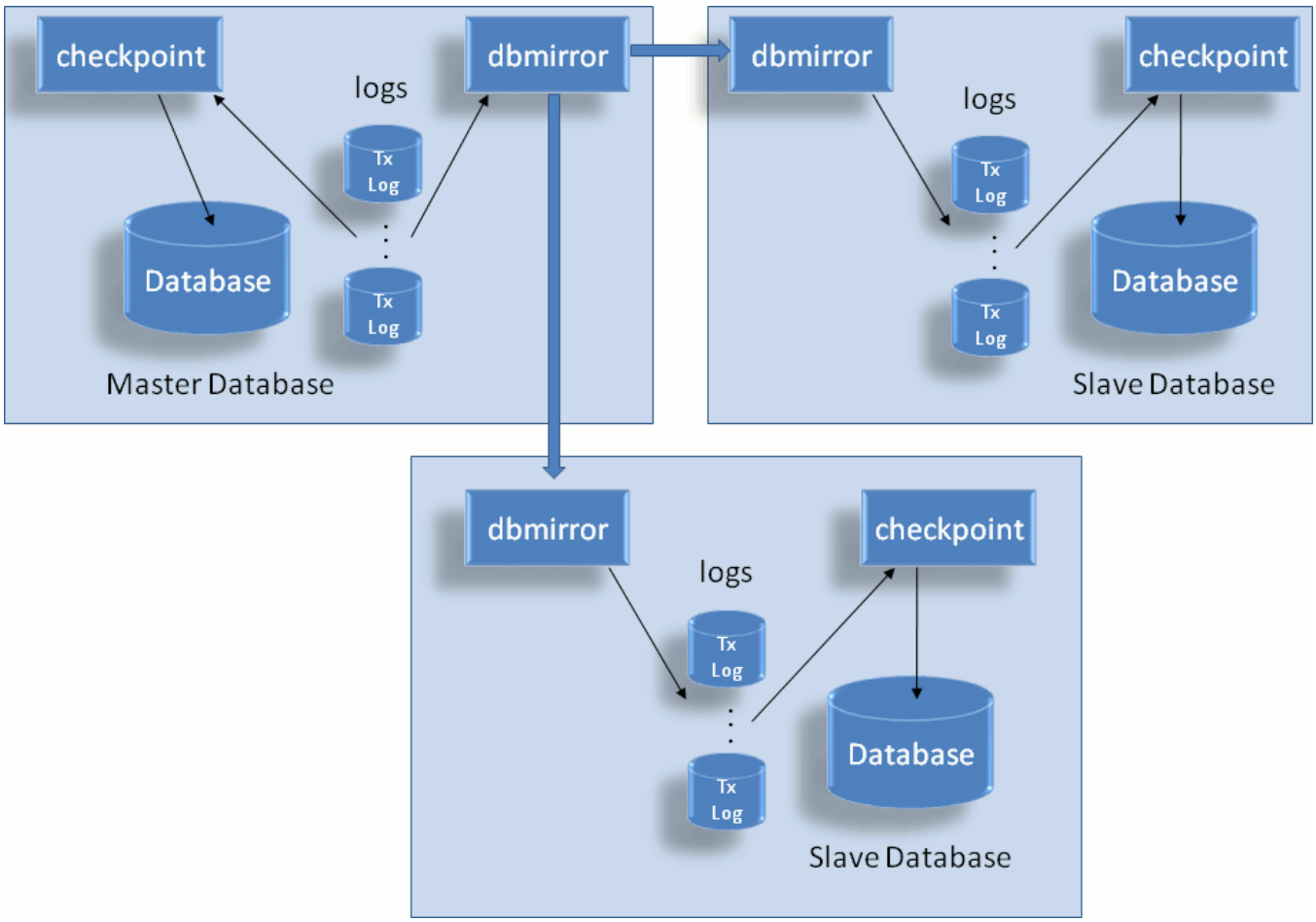

Fig. 12-1 Mirroring Architecture

The mirroring process is as follows:

1. Every transaction is committed to a master database by creating a log file (containing modified page images) in the database's directory. The log files are named after the transaction number they represent. Transactions are numbered serially with no gaps (see discussion about naming below).
2. The checkpoint process scans the directory for log files. When one or more new log files are found, they are "checkpointed" into the database. The entire process is safe and repeatable so that there will be no loss of data.
3. To facilitate mirroring, the checkpoint process will not delete used log files, but will rename them so that they can be found by the `dbmirror` process.
4. The slave `dbmirror` process requests the "next" log from the master `dbmirror` process. When it receives it, it sends it to the local TFS, if a TFS is running, for normal transaction processing. If a TFS is not running together with the slave `dbmirror`, there may or may not be a checkpointing process running. If there is, the presence of the log causes the checkpointer to copy these changes into the slave database. This is repeated forever, or until the dbmirror slave process is terminated.
5. The master `dbmirror` process searches the database directory for log files and responds to slave `dbmirror`s when certain log files are requested.
6. The master `dbmirror` can respond to any number of slave `dbmirror`s.

Certain special conditions exist:

1. When a *first-time request* for a database comes from a slave `dbmirror`, the entire database must be copied to the slave. Then the normal process of applying incremental changes through log files applies.
2. The master checkpointer will have rules by which it deletes or cleans up log files:
    a. If mirroring is disabled, the log files are deleted immediately after they are checkpointed.
    b. If mirroring is enabled, the checkpointer will rename them such that `dbmirror` will still find them, but they will not be checkpointed again.
    c. The checkpointer will be given time and/or disk space rules for log file cleanup. If the log files are beyond a certain age, they will be deleted, or if the total space on disk taken by the log files exceeds a given number, log files will be deleted.
3. If a slave `dbmirror` requests a log that no longer exists (because of rule c above), it will be necessary to treat the request as a first-time request and send the entire database again. This situation can arise when a slave has an intermittent connection, or only connects to "synchronize" the database. There must be a balance between how long a slave may be disconnected and how long the log files will be kept by the master.

Transaction log files are transient. They are kept as long as they are useful. When mirroring is disabled, their usefulness ends as soon as their contents are written to the database. The name of a log file represents it's current state. The list below shows the *life cycle* of a transaction log file:

1. **Creation** - while being created, the log file is opened in the database's directory and named *`dbuserid`*`.prelog`, where `dbuserid` has been automatically assigned by the TFS, or has been requested through the `d_dbuserid` call. Logs being simultaneously created by different runtimes will always have different names.
2. **Commitment** - Once a prelog file has been completely written, it is *synced*, meaning that all of its contents are written and committed to disk. Should the computer fail after this moment, the contents on the disk will be correct and complete. After the sync operation, the file is renamed. The renaming makes it visible to the checkpointing process. If a computer failure occurs after the sync but before the rename, then the transaction will not be considered to be committed, and will never be found in the database after the TFS restarts. In this situation, the runtime submitting the transaction will not be notified that the transaction was successfully committed. The name of the log file will be the transaction ID (an 8 hexadecimal digit number) with a `.log` type, for example, `0000015d.log`. The log files are named strictly sequentially, and are always applied to the database in numeric order.
3. **Checkpoint** - The checkpoint process will find and apply the contents of a log file to the database on a regular basis. Once a log file (and any others that may be batched together) has been written to the database and the database files have been synced, the log files are no longer required by this TFS. If mirroring is disabled, the log file(s) are deleted now. If not, the log files are renamed again by adding a .arch suffix, for example, `0000015d.log.arch`. The dbmirror process looks for transactions in numeric order.
4. **Cleanup** - If archived log files exist, there should be time or space restrictions that determine when they can be deleted.

See Figure 12-2 below:

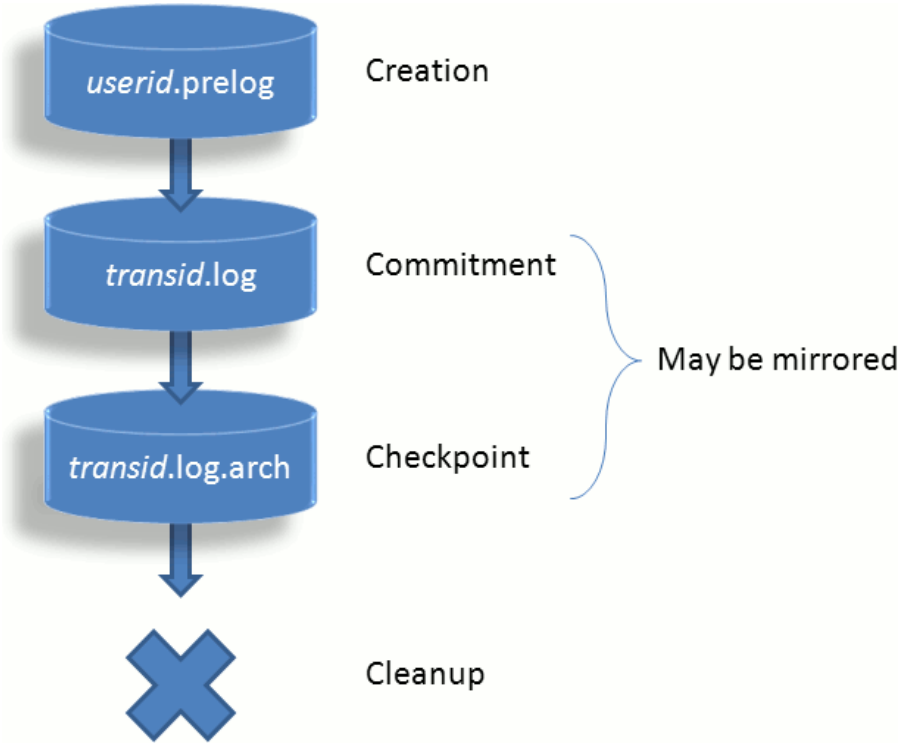

Fig. 12-2 Transaction Log Life Cycle

Replication Process

Figure 12-3 below shows the generalized replication flow.

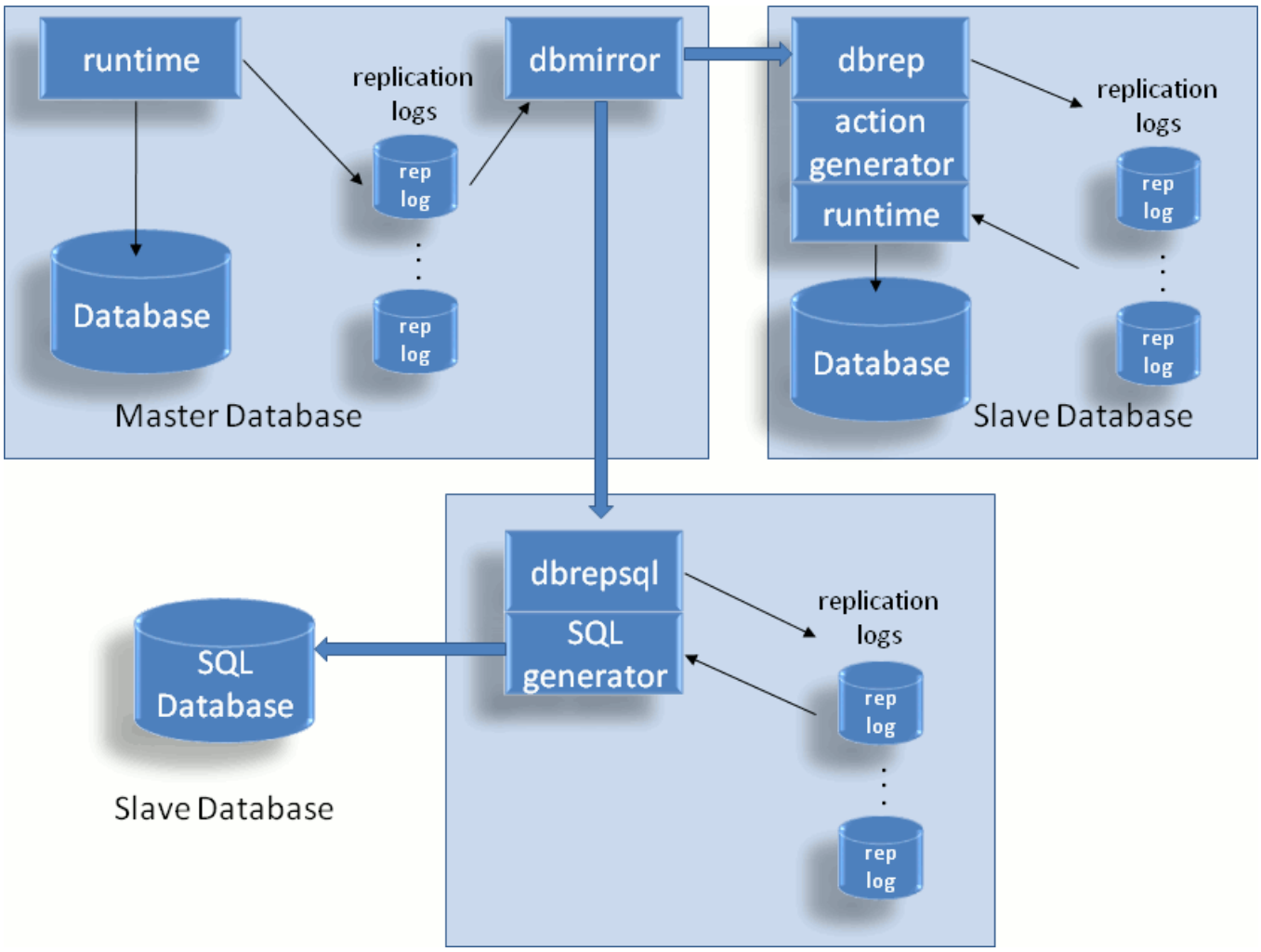

Fig. 12-3 Replication Architecture

Note that figures 12-1 and 12-3 are similar in the master database domain. While all components are not shown in the figures, both use the same dbmirror utility to manage the transmission of log files, whether they are change logs or replication logs. All logs are placed into the same location to be found by dbmirror when they are ready. The process of copying the initial database to a slave is the same. The main difference is how the log file is consumed on the slave side.

Replication preparation is as follows for replication to an *RDM Embedded* database:

1. Start with an existing RDM Embedded database. DDL has been defined and compiled with `ddlp`. This database may be new, or it may be in active operation.

Replication preparation is as follows for replication to an *RDM Server* database:

1. Start with an existing RDM Embedded database. DDL has been defined and compiled with `ddlp`. This database may be new, or it may be in active operation.
2. Create SQL DDL from the existing database definition. See the `schemaxlate` utility definition below. It will generate a text file containing the SQL DDL corresponding to the RDM Embedded database definition.
3. Use the SQL database procedure to generate a new database from the SQL DDL.

4. Create an ODBC data source for the SQL database (Windows only).
5. Set the RDSLOGIN environment variable to identify the ODBC DSN (Windows) or RDM Server name (Unix), username and password.

The replication process is as follows:

1. For every transaction committed to a master database, a replication log is also generated. The log is only generated if a configuration option is turned on. See Mirroring Setup below. The names for the replication log files are also based on the same transaction numbers as the change log files.
2. The slave `dbrep` or `dbrepsql` process requests the "next" log from the master `dbmirror` process. When it receives it, it stores it to the local database directory under the document root.
   a. If `dbrep` is running, the log will be processed into the slave RDM Embedded database.
   b. If `dbrepsql` is running, the log will be converted into SQL and submitted to the SQL database that has been selected.

   This is repeated forever, or until the slave process is terminated.
3. The master `dbmirror` process searches the database directory for log files and responds to slaves when certain log files are requested.
4. The master `dbmirror` can respond to any number of slaves.

Certain special conditions exist:

1. When a *first-time request* for a database comes from a slave process, the entire database must be copied to the slave. Then the normal process of applying incremental changes through log files applies.
2. The master checkpointer will have time and/or disk space rules for log file cleanup. If the log files are beyond a certain age, they will be deleted, or if the total space on disk taken by the log files exceeds a given number, log files will be deleted.
3. If a slave process requests a log that no longer exists (because of rule 2 above), it will be necessary to treat the request as a first-time request and send the entire database again. This situation can arise when a slave has an intermittent connection, or only connects to "synchronize" the database. There must be a balance between how long a slave may be disconnected and how long the log files will be kept by the master.

Database Storage Location

A slave database resides in a slightly different location relative to the TFS document root. All databases mirrored or replicated from the same TFS will be under a subdirectory named after the source TFS and its port. An example is shown below for a mirror site that keeps copies of `sales` and `invntory`:

```
MASTER                                  SLAVE
TFS Name: acct.raima.com                TFS Name: RLM-lptp
Document Root: c:\RDMe\databases        Document Root: d:\db
------------------------                ------------------
c:\RDMe\databases\                      d:\db\
    invntory\                               acct.raima.com-21553\
        invntory.dbd                            invntory\
        invntory.d01                                invntory.dbd
```

```
        ...                                          invntory.d01
    sales\                                           ...
        sales.dbd                                  sales\
        sales.d01                                      sales.dbd
        ...                                            sales.d01
                                                       ...
```

Opening the master database is also different from opening the slave. The open call must include the path to the slave database. Slave databases are read-only and cannot be opened for updating. Assuming that a program is running such that the databases are found on `localhost`:

```
On acct.raima.com                    On RDM-lptp
-----------------                    -----------
d_open("sales;invntory", "s", task);  d_open("acct.raima.com-21553/invntory",
"r", task);
                                     d_iopen("acct.raima.com-21553/sales",
task);
```

Note that throughout this document the notation for identifying any TFS is *`hostname`*`:port`, where `:port` can be omitted if the default port, 21553, is used. We recommend using the default port for everything unless multiple TFSs are running on the same computer. However, when a directory is created for the sake of storing a mirrored or replicated database, the port is not optional. Therefore the default port will be made a part of the directory name even though it may be left off of TFS references. For example, if the `dbget` utility is used to initiate the mirroring of a database as follows:

```
dbget -b poi@tfs.raima.com
```

which is using the default port, then the local directory created under the document root to store the database will be:

```
docroot/tfs.raima.com-21553/poi
```

It is always fine to explicitly include the default port:

```
dbget -b poi@tfs.raima.com:21553
```

Synchronous Mirroring

By default, mirroring is asynchronous. Asynchronous operation is the fastest and most flexible mode for mirroring. Synchronous mirroring forces a program's `d_trend` call to wait until the slave `dbmirror` replies to the master `dbmirror` that the transaction has been committed in a durable manner on the synchronous slave database.

When a synchronous slave database is being used, the implied intent is that the slave database may be used as a primary database should anything happen to the master database, and it is essential that any transaction that has committed in the master can also be found in the slave. Asynchronous mirroring cannot guarantee this.

Because of the usage scenario for synchronous mirroring, once a synchronous mirror database has been established, the master database may not be updated unless the mirror is also being updated. In other words, the slave `dbmirror` utility becomes a required piece in updating the master database. If it is not running, then the master database is read-only. This enforces the notion that once a runtime receives a successful "commit" message, it is committed on both master and slave.

Once a database is being synchronously mirrored, it will remain synchronously mirrored until this mode is deliberately cleared, even if the slave `dbmirror` utility stops and restarts.

Only one slave synchronous mirror may be established. Any other slaves must be asynchronous.

DataFlow Utilities

Four utilities exist to implement DataFlow. The first three, `dbmirror`, `dbrep`, and `dbrepsql` have already been discussed in the section above. Another, `dbget`, is used to initiate the DataFlow process.

The steps below are followed to mirror a database:

1. Run `dbmirror` on the master TFS computer.
2. Run `dbmirror` on the slave computer.
3. Run `dbget` on the slave computer. Identify the master database and the slave `dbmirror`.

The steps below are followed to replicate a database into an RDM Embedded slave:

1. Run `dbmirror` on the master TFS computer.
2. Run `dbrep` on the slave computer.
3. Run `dbget` on the slave computer. Identify the master database and the slave `dbrep`.

The steps below are followed to replicate a database into a SQL database engine:

1. Run `dbmirror` on the master TFS computer.
2. Run `dbrepsql` on the slave computer.
3. Run `dbget` on the slave computer. Identify the master database and the slave `dbrepsql`.

Step 3 is performed for each separate database that is to be mirrored or replicated.

The `dbmirror` utility spawns threads for each database connection. The master `dbmirror`'s threads respond to requests from the slave utilities, which will ask for the *next* transaction log file. The slave `dbmirror` keeps track of the last log file it received. The slave `dbmirror` may also be re-started after a termination or dis-connection. It will determine the transaction ID of the last received log file and begin again (upon request by `dbget`) by requesting the next log file.

The `dbmirror` utility may serve as both master and mirroring slave at the same time. It may be a slave to create a mirror of a database for which it is also the master, for yet another slave requesting the same database. Thus a chain or tree of mirroring can be set up if necessary. It also can serve as master for multiple slaves of the same database, and for multiple databases.

The following examples would set up mirroring from a master database to two slaves using three computers:

The master TFS, `tfs.raima.com`:

```
tfserver -d c:\RDMe\databases
dbmirror -d c:\RDMe\databases
```

The first slave TFS, `RLM-lptp`, requesting the `sales` database:

```
start tfserver -d d:\db
start dbmirror -d d:\db
dbget -b sales@tfs.raima.com
```

The second slave TFS, `acctg-main`, requesting the `sales` and `invntory` databases:

```
cd c:\acct-dbs
start tfserver
start dbmirror
dbget -b sales@tfs.raima.com
dbget -b invntory@tfs.raima.com
```

A single synchronous slave can be started as follows:

```
dbget -sync -b sales@tfs.raima.com
```

To deliberately stop synchronous mirroring, `dbget` must be used:

```
dbget -unsync -b sales@tfs.raima.com
```

4.1 Dbmirror Usage

The `dbmirror` utility is run one time, even when there are two or more master databases that will be mirrored. It will start one thread for each mirrored master database.

The `dbmirror` utility will accompany a TFS which is controlling access to master database(s). If `dbmirror` is to operate as a slave only, the TFS is optional, but when the TFS is not running on a slave computer, only log files will be created in the database subdirectory under the document root. These log files can serve as a backup of a database should the database need to be reconstructed.

The `dbmirror` utility is itself a server which listens on its own separate port. However, when starting or referring to this utility, the *anchor port* (the port the TFS is using) is used. For those who need to make sure the proper ports are open in a firewall, the utility's port is the anchor port plus 1.

```
dbmirror [-d PATH] [-p N] [-v] [-nodisk]

  -d               = PATH location of server document root (absolute, or relative
                     to current directory)
  -p               = TCP Anchor Port N of slave's TFS (default is 21553)
  -v               = Verbose output
  -nodisk          = do not store log files on the local disk drive
  -stdout filename = Specify a file name to write errors and warnings
```

The `-d` option should identify the same document root directory as the one being used by the TFS. If this option is not specified, the current directory is used.

The `-v` option will print a log of mirroring activity. Use this to verify that your configuration is working, but leave it off for normal operation because it will interfere with performance.

The `-nodisk` option is used when databases are defined to be inmemory and you want the DataFlow utility to keep its files only in memory also.

4.2 Dbrep, dbrepsql Usage

The `dbrep` or `dbrepsql` utilities are servers which listen on their own separate ports. However, when starting or referring to one of them, the *anchor port* (the port the TFS is using) is used. For those who need to make sure the proper ports are open in a firewall, the utility's port is the anchor port plus 1. For this reason, you cannot run both `dbrep` and `dbrepsql` with the same anchor port.

```
{dbrep|dbrepsql} [-d PATH] [-p N] [-v] [-nodisk]

  -d               = PATH location of server document root (absolute, or relative
                     to current directory)
  -p               = TCP Anchor Port N of slave's TFS (default is 21553)
  -v               = Verbose output
  -nodisk          = do not store log files on the local disk drive
  -stdout filename = Specify a file name to write errors and warnings
```

The command-line options for both utilities are the same, and the definitions are the same as in the `dbmirror` utility. For `dbrep`, a local RDM Embedded database will be created and/or maintained. For `dbrepsql`, an SQL database matching the definition of the master RDM Embedded database must have been defined and made accessible. See the `schemaxlate` utility and Replication Setup sections for more details.

The `dbrep` or `dbrepsql` utility is run one time, even when there are 2 or more databases that will be replicated. It will start one thread for each replicated database.

The `-nodisk` option is used when databases are defined to be inmemory and you want the DataFlow utility to keep its files only in memory also.

4.3 Installation as Service or Daemon Process

The following options are available for the `dbmirror`, `dbrep` or `dbrepsql` utilities for the purpose of starting them in the background:

```
{dbmirror|dbrep|dbrepsql} [-start|-stop|-query] [-stdout filename]

  -start = Start this utility as a background process
  -stop  = Shut down the utility
  -query = Determine if the utility is running in the background or not
```

The `-stdout` *`filename`* option is used when stdout is not appropriate, such as when the utility is started in the background. All error and warning output will be written to *`filename`*.

The following options function on Windows systems in order to treat the utility as an automatic Windows service:

```
{dbmirror|dbrep|dbrepsql} [-install exepath|-uninstall]

  -install exepath = Install utility as a service. The exepath is the directory
                     containing the utility, or directory\utility.EXE
  -uninstall       = Uninstall this utility
```

When installed as a service, use the Windows Services Manager to start and stop, rather than the command-line options above. By default, the service will be automatic, and will be started when it is installed.

4.3 Dbget Usage

The `dbget` utility is used to make a request to a slave process (`dbmirror`, `dbrep` or `dbrepsql`) to initiate mirroring or replication for a particular database. The slave process must be running before `dbget`'s notification can be processed. For every database being mirrored or replicated, `dbget` must be invoked once. `Dbget` is also used to cleanly stop mirroring or replication. Mirroring or replication may be started up again after it has been stopped.

```
dbget [-u DBUSERID] [-s HOSTNAME] [-p N] [-sync] [-unsync] [-b] [-e] [-override_
inmem]
      [-dsn dsn;user;pswd] [-oracle|-mssql|-mysql] dbname[@m-
asterTFSdomain][:masterTFSport]

  -u                                        = DBUSERID to use during
                                              transactions
  -s                                        = HOSTNAME of slave DBMIRROR
                                              (default is localhost)
  -p                                        = Anchor port N of the slave DBMIRROR
                                              (default is 21553)
  -sync                                     = Mirrored database is synchronous
  -unsync                                   = End persistent synchronous
                                              mirroring
  -b                                        = Begin mirroring or replication
  -e                                        = End mirroring or replication
  -override_inmem                           = This slave database is on-disk,
                                              regardless of the master's storage
media
  -dsn                                      = Specify a DSN for dbrepsql
  -oracle                                   = Slave dbrepsql connects to Oracle
                                              server
  -mssql                                    = Slave dbrepsql connects to
                                              Microsoft SQL server
  -mysql                                    = Slave dbrepsql connects to MySQL
                                              server
  dbname[@masterTFSdomain][:masterTFSport] = Name and location of master
                                              database TFS.
                                              Default domain is localhost.
                                              Default port is 21553.
```

Together, the `-s` and `-p` options identify the location of the slave utility that will "get" a database. When not specified, `localhost:21553` is used, which are the defaults used by the `dbmirror` utility.

It is not necessary to use `-u` to provide a DBUSERID. This option may be specified in order to distinguish this utility from other runtimes when monitoring system activity.

The `-sync` and `-unsync` options place a slave mirror database into or out of synchronous mirroring mode, respectively. If another synchronous mirror for this database already exists, the `-sync` request will be rejected.

4.4 Schemaxlate Usage

```
schemaxlate [-n] [-f] [-o outfile] [-d device] [-oracle|-mssql|-mysql] dbname

  -n      = Force fields to allow NULLs
  -f      = Force overwrite of existing output file
  -o      = Name of output file, default is <dbname>_rdms.sql
  -d      = Name of device for files, default is sqldev (for RDMs)
  -oracle = Create output compatible with Oracle (default output file is
            <dbname>_oracle.sql)
  -mssql  = Create output compatible with Microsoft SQL Server (default output
            file is <dbname>_mssql.sql)
  -mysql  = Create output compatible with MySQL (default output
            file is <dbname>_mysql.sql)
```

By default, this utility generates SQL DDL for RDM Server. The options for Oracle, MS SQL Server or MySQL cause slight variations in the DDL to make it compatible with the target system.

Mirroring Setup

Mirroring is enabled by an INI file setting. In the '[configuration]' section, the key 'mirroring' is set to 1 to enable or 0 to disable mirroring.

The `ini` files may be at the server or database level. At the server level, the document root may contain a file named `tfs.ini` that may contain any of the below-defined parameters. If specified, the parameter values are the default values for all databases within the scope of this TFS. The default values may be overridden by placing *`dbname`*`.ini` into the database's subdirectory under the document root.

```
[configuration]
; mirroring is set to 1 (TRUE) to indicate that a database will be
; mirrored. This option is not required for and does not affect replication.
; When set to 1, change log files will not be immediately deleted.
; Default is 0.
mirroring=1
```

By default, the `tfserver` utility will delete change log files after they have been checkpointed to the database. To mirror a database, these log files must be retained. Log retention is controlled through two `ini` file parameters, as follows:

```
[LogCleanup]
; LogFileAge gives amount of time to retain log files before removing them.
; Use Ns for N seconds, m for minutes, h for hours, or d for days.
; Default is 10m.  Set to 0 to never remove logs.
LogFileAge=10m
; LogFileSpace gives disk space used by log files before removing them.
; Use Nk for N kilobytes, m for megabytes, or g for gigabytes.
; There is no default space limit.  Also, set to 0 for no space limit.
LogFileSpace=100m
```

When log files are removed, the oldest are always removed first. This may trigger a full-database refresh if a mirroring or replicating client makes a request for a *next* log that has been deleted.

Default Options with Masters and Slaves on Different Computers

When only one TFS is run on each computer, the default ports should be used, and may be left off the command-line. The default port for `tfserver` is `21553`. The `dbget` utility refers to anchor port `21553` to connect to `dbmirror`. The following example assumes that two Windows and two Linux machines are being used.

Master 1 (Windows, `tfs.raima.com`), controlling `sales` and `mkt` databases:

```
c:
cd \RDMe\databases.win32
start tfserver
start dbmirror
```

Master 2 (Linux, `RLM-lptp`), controlling `tims` database:

```
cd /RDMe/databases.lnx
tfserver &
dbmirror &
```

Slave 1 (Windows, `WLW-XP`):

```
d:
cd \databases.win32
start tfserver
start dbmirror
dbget -b mkt@tfs.raima.com
dbget -b sales@tfs.raima.com
```

Slave 2 (Linux, `dai-desktop`):

```
tfserver -d /home/databases &
dbmirror -d /home/databases &
dbget -b mkt@tfs.raima.com
dbget -b tims@RLM-lptp
```

Using dbget From a Different Computer

The `dbget` utility will commonly be used from within the context of the slave database, but it doesn't need to be. It simply needs to address the correct `dbmirror` slave process. Assuming that the master and slave processes have been set up as above, then the following `dbget` invocations will achieve the same results and will work from any of the computers:

```
dbget -b -s WLW-XP mkt@tfs.raima.com
dbget -b -s WLW-XP sales@tfs.raima.com
dbget -b -s dai-desktop mkt@tfs.raima.com
dbget -b -s dai-desktop tims@RLM-lptp
```

Non-Default Options with Masters and Slaves on the Same Computer

This example shows the opposite extreme from the first example, where all masters and slaves are now to operate on the same computer. We will not use default ports for anything, although they will all use `localhost` as the domain. All `tfserver` and `dbmirror` processes must use different ports, and the `dbget` utility must refer to the correct ports. The following table shows how this will be configured:

Role	Utility	Document Root	Port
Master 1	tfserver	c:\RDMe\databases.win32	1730
Master 1	dbmirror	c:\RDMe\databases.win32	1730

Role	Utility	Document Root	Port
Master 2	tfserver	c:\databases	1830
Master 2	dbmirror	c:\databases	1830
Slave 1	tfserver	d:\databases.win32	1840
Slave 1	dbmirror	d:\databases.win32	1840
Slave 2	tfserver	d:\home\databases	1940
Slave 2	dbmirror	d:\home\databases	1940

Table 12-1 Same-computer Mirroring Configuration

Master 1, controlling `sales` and `mkt` databases:

```
c:
cd \RDMe\databases.win32
start tfserver -p 1730
start dbmirror -p 1730
```

Master 2, controlling `tims` database:

```
c:
cd \databases
start tfserver -p 1830
start dbmirror -p 1830
```

Slave 1:

```
d:
cd \databases.win32
start tfserver -p 1840
start dbmirror -p 1840
dbget -b -p 1840 mkt@localhost:1730
dbget -b -p 1840 sales@localhost:1730
```

Slave 2:

```
start tfserver -p 1940 -d d:\home\databases
start dbmirror -p 1940 -d d:\home\databases
dbget -b -p 1940 mkt@localhost:1730
dbget -b -p 1940 tims@localhost:1830
```

Using dbget From a Different Computer, Again

This example corresponds to the example above, where all of the utilities are being run on the same computer (say, `tfs.raima.com`). The `dbget` utility can be run from anywhere, and in this case, it must specify the correct ports (for the `dbmirror` process and master database) as well as the domain:

```
dbget -b -s tfs.raima.com -p 1840 mkt@tfs.raima.com:1730
dbget -b -s tfs.raima.com -p 1840 sales@tfs.raima.com:1730
dbget -b -s tfs.raima.com -p 1940 mkt@tfs.raima.com:1730
dbget -b -s tfs.raima.com -p 1940 tims@tfs.raima.com:1830
```

Replication Setup

Slave databases are replicated, rather than mirrored, when the `dbrep` or `dbrepsql` utilities are used rather than the `dbmirror` utility for receiving and updating the slave database. The master database always uses `dbmirror` for handling both mirrored and replicated slaves.

Replication is enabled by an INI file setting. In the `[configuration]` section, the key `replication` is set to `1` to enable replication. Remove this key or set to `0` to disable replication.

```
[configuration]
; replication is set to 1 (TRUE) so that replication log files will be generated.
; The 'mirroring' parameter is not required for replication.
; Default is for no generation of replication log files.
replication=1
```

Default Options with Masters and Slaves on Different Computers

When only one TFS is run on each computer, the default ports should be used, and may be left off the command-line. The default port for `tfserver` is `21553`. The `dbget` utility refers to anchor port `21553` to connect to `dbrep`. The following example assumes that two Windows and two Linux machines are being used.

Master 1 (Windows, `tfs.raima.com`), controlling `sales` and `mkt` databases:

```
c:
cd \RDMe\databases.win32
start tfserver
start dbmirror
```

Master 2 (Linux, `RLM-lptp`), controlling `tims` database:

```
cd /RDMe/databases.lnx
tfserver &
dbmirror &
```

Slave 1 (Windows, `WLW-XP`) replicating to RDM Embedded slaves:

```
d:
cd \databases.win32
start tfserver
start dbrep
dbget -b mkt@tfs.raima.com
dbget -b sales@tfs.raima.com
```

Slave 2 (Linux, `dai-desktop`) replicating to RDM Server slaves:

```
export RDSLOGIN="RDMs;admin;secret"
tfserver -d /home/databases &
dbrepsql -d /home/databases &
dbget -b mkt@tfs.raima.com
dbget -b tims@RLM-lptp
```

Using dbget From a Different Computer

The `dbget` utility will commonly be used from within the context of the slave database, but it doesn't need to be. It simply needs to address the correct `dbrep` slave process and database. Assuming that the master and slave processes have been set up as above, then the following `dbget` invocations will achieve the same results and will work from any of the computers:

```
dbget -b -s WLW-XP mkt@tfs.raima.com
dbget -b -s WLW-XP sales@tfs.raima.com
dbget -b -s dai-desktop mkt@tfs.raima.com
dbget -b -s dai-desktop tims@RLM-lptp
```

Non-Default Options with Masters and Slaves on the Same Computer

Refer to the corresponding section in Section 12.5 above. The difference between replication and mirroring is in the use of `dbrep` or `dbrepsql` in place of the slave `dbmirror`.

Opening Slave Databases from Programs

The `d_open` or `d_iopen` functions are able to open a database mirror for reading. Referring to the example above, an application running on `RLM-lptp` can open the database mirrored from `tfs.raima.com` with the following statement:

```
d_open("tfs.raima.com-21553/sales", "r", task);
```

Or, if the application is not necessarily running on `RLM-lptp` (although it may be), the following reference to the sales database will work from anywhere:

```
d_open("tfs.raima.com-21553/sales@RLM-lptp.raima.com", "r", task);
```

Generally, the advantage of opening a mirror for reading is because it is local and therefore faster.

Another reason to mirror a database to the local computer is to view it in a union with other identically structured databases. For example, suppose that `acctg-main` needs to create a report on `invntory`, which is maintained independently at two sites. The first is on `tfs.raima.com` (as in the example above), and the second is on `sf.raima.uk`. A mirror of `invntory` from both sites is mirrored to the local computer. Then the following statement will be able to open a merged view of the inventory:

```
d_open("tfs.raima.com-21553/invntory|sf.raima.uk-21553/invntory", "r", task);
```

Advanced Topics

8.1 Differences Between Master and Slave

Rules for creating safe differences between the master and slave databases are discussed in the following sections. All techniques involve creating a database dictionary (DBD) file that is different, but compatible between the master and slave.

Frequently, when a slave database is created, its location will be empty, and the DBD from the master will be used. But once a DBD file has been created and placed into the slave database location, the DBD will not be overwritten by future copies of the database. All of the techniques described below depend on the ability to create a slave DBD that will not be overwritten by the master DBD.

8.2 In-Memory to On-Disk

A common reason to mirror or replicate is to create a permanent, safe copy of a database while maintaining an in-memory master database for performance reasons.

To create a slave database that is on-disk, regardless of the storage media of the master database (either in-memory or on-disk), use the `-override_inmem` option to `dbget`. **This will take effect for either mirroring or replication** (where replication is to an RDM Embedded slave). When this option is specified, the initial copy of the master database to the slave location will also alter the DBD such that the slave database is marked as on-disk. Subsequent copies of the database (because logs get out of range) will not cause this DBD to be overwritten, so the condition is permanent unless the slave database is completely destroyed.

Note that if additional changes must be made, the `-override_inmem` option will not effect this change - it must be made part of the original slave DDL as shown below.

8.3 Replication-Only Changes for RDM Embedded Databases

The only change available to mirrored databases is the storage media. Replication offers more flexibility because changes are propagated from master to slave through replaying a series of changes, not through copying page images. It is possible to replay changes into a slave database without disturbing other existing pieces of the database, also allowing some differences in the results of the replayed changes.

As an example of a difference in replay results, consider a master database that quickly stores incoming data as records with no keys. The slave database can be defined to have keys that do not exist in the master database. When a `d_fillnew` is called to create a record in the master database, only the record is stored, but when the same `d_fillnew` with the same data fields is replayed into the slave database, one or more keys can be created by the call. An example of differences between DDL's is shown below:

Master/Slave DDL snippets

```
data file "station.d01" contains sta-
tiondata;

record stationdata {
    int32_t dayOfYear;
    int32_t minOfDay;
    int32_t totalPrecip;
    int32_t windDir;
    int32_t windVel;
    int32_t temp;
}
```

```
data file "station.d01" contains sta-
tiondata;
key file "station.k01" contains minOfDay;
key file "station.k02" contains timeKey;

record stationdata {
      int32_t dayOfYear;
  key int32_t minOfDay;
      int32_t totalPrecip;
      int32_t windDir;
      int32_t windVel;
      int32_t temp;
      compound key timeKey {
          dayOfYear;
          minOfDay;
      }
}
```

In this example, simple records are created in the master, but two keys are defined in the slave. This example is correct because the record type will have the same ID (that is, no new record types were defined before it), the record has the same original data fields in the same order, and the files into which the keys will go are also new, leaving the existing file with the same ID. This follows rule 1 for safe changes to DDL:

Rule 1 - Safe Changes to DDL

The slave DDL must contain *all* original files, record types and fields *in the original order*. Any new record types must follow all existing record types. No new fields may be added. Keys may be added. File definitions must be added following all existing file definitions. All new record types and keys must be placed into new files - they cannot be added to existing files.

Rule 2 - Restrictions on Unique Keys

If a unique key exists in the slave DDL, it must also exist in the master DDL. This is because it is necessary to make sure that no duplicate unique keys are created on the slave, which will cause the replication of the record to fail.

Non-unique keys may be included in the slave DDL where the master DDL has no key, a non-unique key, or a unique key.

Rule 3 - Changing the Storage Media

As mentioned above, the `-override_inmem` option in `dbget` cannot be used if other changes, shown in this section, are also being made. However, the slave DDL may be modified to change the storage media, especially from in-memory to on-disk, as shown below:

Master/Slave DDL snippets

```
database weatherdata inmemory {

    ...
```

```
database weatherdata {

    ...
```

```
}
```

```
}
```

Rule 4 - Circular Tables

Circular tables should not be replicated to a slave database. Instead, it is possible to define the slave table (record type) as a normal record type, which becomes a permanent log that captures and keeps each record that was entered into the master's circular table. The `circular` keyword should be removed, as should the `maxslots` definition, as shown below:

Master/Slave DDL snippets

```
data file "turnstile.d01" maxslots=300
          contains turnstile;
    ...

circular record turnstile {
    char    ticketCode[21];
    int32_t clock;
    int32_t turnstilenum;
}
```

```
data file "turnstile.d01" contains turn-
stile;
key file "turnstile.k01" contains
seqKey;
    ...

record turnstile {
    char    ticketCode[21];
    int32_t clock;
    int32_t turnstilenum;
    compound key seqKey {
        turnstilenum;
        clock;
    }
}
```

Note that the above example also shows a new key that can be used to maintain some ordering on the records. Records in the slave database will be created in the same order as in the master database, meaning that the original sequence can be followed by using the `d_recfrst`, `d_recnext` functions.

8.4 Replication-Only Changes for SQL Databases

The first step when replicating to a SQL database is to generate SQL DDL to define tables and columns in the SQL database the correspond to the original RDM Embedded record and field types. While this can be created manually, it is highly recommended that you use the `schemaxlate` utility. This will provide a minimal starting point even if you plan to modify the SQL DDL produced by the utility.

The output of schemaxlate may be used to define a new database, or it may be added to an existing database.

Rule 1 - All Record and Field Types in Master Must Exist in Slave

Whether creating SQL DDL manually, or modifying the output of `schemaxlate`, the replication mechanisms assume that any modifications made in the master can also be made in the slave.

The remaining rules assume that Rule 1 is being followed.

SQL slave databases allow for more flexibility than RDM Embedded slave databases when changes from the original RDM Embedded master database are desired. All SQL updates are created by converting the

replication log files into SQL statements. For example, when `d_fillnew` is used to create a new record in the master database, a replication log entry is created that identifies the `d_fillnew` function and the structure containing the field values. When this replication log is interpreted by the `dbrepsql` utility, it is converted into:

```
INSERT INTO recordtypename (fldname1, fldname2, ...) VALUES (val1, val2, ...)
```

where the *recordtypename* and *fldname*s match those of the original record type.

Since SQL replication is performed by table and column names, there is no need to make sure that record, field and file ID's match up on both master and slave. This leads to rule 2:

Rule 2 - Safe Changes to SQL DDL

New tables may be added, and it is unimportant whether they precede or follow existing tables. Columns (with NULL values allowed) may be added to tables, the order is not important. Keys may be added, but they must not be unique or primary keys. Each row will already have a primary key defined.

Rule 3 - Referential Integrity

You may not add foreign keys to any existing tables, to avoid errors that occur if the primary key does not (yet) exist.

Rule 4 - Circular Tables

The SQL DDL created by `schemaxlate` will have a normal table in the place where any circular table existed. There is nothing that needs to be done to make this work correctly.

8.5 Slave Database Setup

This section summarizes the steps needed to initiate and maintain mirroring or replication in its various permutations. Locate your intended usage in the table and refer to the setup section indicated.

	Mirrored RDMe Slave	Replicated RDMe Slave	Replicated ODBC SQL Slave
No DDL Change	1 - Normal Mirroring	2 - Normal Replication	3 - Normal SQL Replication
Storage Media Change	4 - Override In-memory	5 - Override In-memory	3 - Normal SQL Replication
DDL Changes	N/A	6 - Advanced Slave Setup	7 - Advanced SQL Replication

Setup 1 - Normal Mirroring, RDMe Slave

Assuming that the master TFS is running on `tfs.raima.com:1730` and the master Mirroring Utility is referencing the TFS at `tfs.raima.com:1730`, normal mirroring setup of the `mkt` database in the slave environment is as follows:

1. Start TFS.

```
tfserver -d /users/RDMe/databases
```

2. Start Mirroring utility.

```
dbmirror -d /users/RDMe/databases
```

3. Initiate Mirroring.

```
dbget -b mkt@tfs.raima.com:1730
```

Setup 2 - Normal Replication, RDMe Slave

Assuming that the master TFS is running on `tfs.raima.com:1730` and the master Mirroring Utility is referencing the TFS at `tfs.raima.com:1730`, normal replication setup of the `mkt` database in the slave environment is as follows:

1. Start TFS.

```
tfserver -d /users/RDMe/databases
```

2. Start Replication utility.

```
dbrep -d /users/RDMe/databases
```

3. Initiate Replication.

```
dbget -b mkt@tfs.raima.com:1730
```

Setup 3 - Normal SQL Replication, ODBC SQL Slave

Assuming that the master TFS is running on `tfs.raima.com:1730` and the master Mirroring Utility is referencing the TFS at `tfs.raima.com:1730`, normal replication setup of the `mkt` database in the slave environment is as follows:

1. Create the SQL DDL.

```
schemaxlate -mysql mkt@tfs.raima.com:1730
```

2. Start Replication utility. Set the environment variable RDSLOGIN to point to the ODBC Data Source, username and password.

```
set RDSLOGIN=mysql-dsn;myname;mypw
dbrepsql -d \users\RDMe\databases
```

3. Initiate Replication.

```
dbget -b mkt@tfs.raima.com:1730
```

Setup 4 - Mirroring, Override In-Memory, RDMe Slave

Assuming that the master TFS is running on `tfs.raima.com:1730`, the master database is in-memory, and the master Mirroring Utility is referencing the TFS at `tfs.raima.com:1730`, mirroring setup of the `mkt` database in the slave environment is as follows:

1. Start TFS.

```
tfserver -d /users/RDMe/databases
```

2. Start Mirroring utility.

```
dbmirror -d /users/RDMe/databases
```

3. Initiate Mirroring. Add the command-line option to make slave store database on disk.

```
dbget -override_inmem -b mkt@tfs.raima.com:1730
```

Setup 5 - Replication, Override In-Memory, RDMe Slave

Assuming that the master TFS is running on `tfs.raima.com:1730`, the master database is in-memory, and the master Mirroring Utility is referencing the TFS at `tfs.raima.com:1730`, replication setup of the `mkt` database in the slave environment is as follows:

1. Start TFS.

```
tfserver -d /users/RDMe/databases
```

2. Start Replication utility.

```
dbrep -d /users/RDMe/databases
```

3. Initiate Replication. Add the command-line option to make slave store database on disk.

```
dbget -override_inmem -b mkt@tfs.raima.com:1730
```

Setup 6 - Replication, Advanced Slave Setup, RDMe Slave

Assuming that the master TFS is running on `tfs.raima.com:1730` and the master Mirroring Utility is referencing the TFS at `tfs.raima.com:1730`, replication setup of the `mkt` database in the slave environment is as follows:

1. Start TFS.

```
tfserver -d /users/RDMe/databases
```

2. Edit the DDL, creating new file, `mkt-slave.ddl`, following rules outlined above.

3. Compile the DDL into the slave location. The ddlp utility must know the location of the master's TFS in order to create the correct location for the slave database.

```
ddlp -master tfs.raima.com:1730 mkt-slave.ddl
```

2. Start Replication utility.

```
dbrep -d /users/RDMe/databases
```

3. Initiate Replication.

```
dbget -b mkt@tfs.raima.com:1730
```

Setup 7 - Advanced SQL Replication, ODBC SQL Slave

Assuming that the master TFS is running on `tfs.raima.com:1730` and the master Mirroring Utility is referencing the TFS at `tfs.raima.com:1730`, normal replication setup of the `mkt` database in the slave environment is as follows:

1. Create the SQL DDL.

```
schemaxlate -mysql mkt@tfs.raima.com:1730
```

2. Edit the SQL DDL file, mkt_mysql.sql, following rules outlined above.

3. Start Replication utility. Set the environment variable RDSLOGIN to point to the ODBC Datasource, username and password.

```
set RDSLOGIN=mysql-dsn;myname;mypw
dbrepsql -d \users\RDMe\databases
```

4. Initiate Replication.

```
dbget -b mysql mkt@tfs.raima.com:1730
```

8.6 Synchronization Issues

The data flow utilities will fetch entire databases when the master and slave are not able to synchronize. During the initial creation of a slave database, the database is copied from the master to establish a starting point, followed by the application of log files (page image or replication) to keep the slave current. This section discusses what is going on when the attempts to remain current fail for some reason.

All three slave data flow utilities keep track of the last log file that they received and applied to the slave database. The operation of the slave utilities is to continuously request the "next" log from the master. If the "next" log is not available, the utility must refresh the database. There are differences in the way this refresh occurs, which will be described below.

The master database directory stores all mirroring or replication logs which are available for request by slaves. Because it is not practical to store all log files for all time, there are configurable limits on the age of log files, or the total storage space taken by log files. When they are removed, they are always removed from the oldest forward.

There are just a few reasons why slave databases get "out of sync:"

1. The slave has been disconnected for a period of time such that upon reconnection, the next log in the sequence has already been cleaned up by the master.
2. The slave is unable to keep up with the updates made by the master, so that even though the slave is connected, it is so far behind the master that the master has started deleting logs still needed by the slave.
3. For some reason, the slave's database directory has been "cleaned up" manually, destroying the records of which log is the next log. (Manual maintenance of the database directories is not recommended).
4. Manual cleanup of the logs in the master database can also create an out-of-sync condition.

Synchronization and Mirroring

The action taken by the slave `dbmirror` utility whenever it cannot obtain the "next" log file is always the same as the initial copy of the database. It will learn that the log file it is requesting is not available, so it will begin requesting pieces of the database instead. The database that is transferred to the slave is stamped with the ID of the last transaction that was applied to it, so following the successful copy of the database, the slave requests the log for transaction ID+1.

Synchronization and Replication

When replication is used to update an RDM Embedded database (non SQL), the action is identical as mirroring, discussed above, with the exception of circular tables. When a circular table is replicated to a continuous table,

this table will not be overwritten if the database needs to be re-copied to the slave. This means that the table will continue to grow rather than re-starting at the size of the existing circular table.

To synchronize a SQL database, `dbrepsql` makes a complete copy of the database in the slave environment, then after some dependency analysis proceeds to DELETE the out-of-sync tables from the SQL database (with the exception of circular tables, which will not be deleted from the slave database). After the deletes have been completed, it scans the replication logs for records, each one of which will be turned into an INSERT statement.

Once the SQL database has been (re)populated, the copy of the RDM Embedded database on the slave is obsolete. Ongoing updates to the SQL database require only new replication log files.

Because circular tables are converted into continuous tables that are not restarted if the database is re-copied from the master, it is possible that gaps may exist in this table.

Balance

Some distributed database applications may be designed to be intermittently synchronized. For example, a corporate contact list may have very few changes to it (perhaps 10 per week). If an employee maintains a slave of this database on their laptop, then it may be necessary to connect to the master every week or two. It will be easy to configure the master database to clean up log files that are two months old.

Another example, like the Market example, involves a database that has repeated changes to the same records. Very quickly, the size of the database can be overshadowed by the size of the change logs. At a certain point, it makes more sense to copy the entire database to slaves rather than copy all of the change logs. There is no formula for this, but the controls are the LogFileAge and LogFileSpace parameters.

Another balance issue is the relative speed of the master updates vs. the slave consumption of the updates. This system is made to handle bursts of updates on the master with pauses that allow the slave(s) to catch up. If the master perpetually outpaces the slaves, it may also create a cycle of re-copying the entire database, which just makes matters worse. If system testing reveals that the master log files are being produced faster than the slave(s) can consume them, it will be necessary to:

1. Reduce the number of slaves,
2. Speed up the slave computers, or
3. Speed up the connection between master and slaves.

If one of the slaves is a synchronous mirror, this can also slow down the consumption of log files significantly.

DataFlow and Mirroring API Functions

The functions beginning with "d_db" are used to control the execution of the DataFlow utilities as in-process functions or threads. The DataFlow utilities include dbmirror, dbrep, and dbrepsql. The utilities are initialized through the d_dbmir_init, d_dbrep_init or d_dbrepsql_init functions, respectively. All three utilities are controlled by the d_dbrep_start, d_dbrep_stop and d_dbrep_term functions. The dbget utility initiates and terminates mirroring or replication connections between master and slave DataFlow utilities. It is implemented through the two functions d_dbrep_connect and d_dbrep_disconnect.

Function	Description
d_dbmir_init	Provide the mirroring utility with parameters to start up as a server
d_dbrep_connect	Initiate a mirroring or replication connection between master and slave (dbget -b).
d_dbrep_disconnect	Terminate a mirroring or replication connection between master and slave (dbget -e).
d_dbrep_init	Provide the mirroring utility with parameters to start up as a server
d_dbrep_start	Begin execution of an initialized DataFlow utility
d_dbrep_stop	Terminate DataFlow utility thread
d_dbrep_term	Clean up resources allocated by DataFlow utility initialization
d_dbrepsql_init	Provide the mirroring utility with parameters to start up as a server

d_dbmir_init

Provide the mirroring utility with parameters to start up as a server

Prototype

```
int32_t d_dbmir_init(
    const DBREP_INIT_PARAMS *rparams,
    REP_HANDLE              *hREP);
```

```
int32_t d_dbmir_initW(
    const DBREP_INIT_PARAMSW *rparamsW,  /* structure with UNICODE string */
    REP_HANDLE               *hREP);
```

Parameters

rparams	(input)	Parameters required for dbmirror operation.
hREP	(output)	Handle used for running, stopping or terminating this server.

Description

This function initializes the functionality of the mirroring utility so that it can operate as a server, and provides a handle to control the utility.

When d_dbmir_init has been called, then d_dbrep_term should be called prior to termination of the application program.

Once this function has been called, the mirroring utility can be controlled through the functions d_dbrep_start, d_dbrep_stop, and d_dbrep_term.

DBREP_INIT_PARAMS structure definition

Element	Declaration	Description
docroot	char/wchar_t	Document root - should match that of the corresponding TFS.
port	uint16_t	TFS anchor port. Should match TFS -p port number (use 21553 if default).
verbose	uint16_t	Verbose, true or false.
diskless	uint16_t	Diskless, true or false.
stdout_file	const char *	File name if stdout is to be redirected to a file, else set to PSP_STDOUT

TFS User's Guide

Direct-Link Configuration

Required Headers

```
#include "rdm.h"
#include "mirutils.h"
```

Libraries

Library Name	Description
rdmerdm10	RDMe Runtime Library
rdmedbmirror10	Mirroring/Replication Control Functions Library
rdmedbmir10	Mirroring Initialization Library

See Library Naming Conventions section for full library name and a list of library dependencies.

Required Packages

High Availability , or

DataFlow

Return Codes

Value	Name	Description
0	S_OKAY	A normal return; everything worked successfully
-904	S_NOMEMORY	The system is unable to allocate sufficient memory.
-924	S_INVNULL	Invalid NULL parameter.

See Also

d_dbrep_init

d_dbrepsql_init

d_dbrep_start

d_dbrep_stop

d_dbrep_term

Example

```
int main(int argc, char *argv[])
{
    DBREP_INIT_PARAMS  rparams;
```

```
    REP_HANDLE          hREP;
    uint16_t            rep_done;
    int32_t             rc;

    memset(&rparams, 0, sizeof(DBREP_PARAMS));

    rparams.docroot      = "c:\\RDMe\\databases";
    rparams.port         = 21553;
    rparams.stdout_file  = PSP_STDOUT;

    rc = d_dbmir_init(&rparams, &hREP);

    if (rc == S_OKAY)
    {
        /* Starts mirroring server code to process requests */
        d_dbrep_start(hREP, TRUE, &rep_done);

        /* wait for termination conditions */
        while( !rep_done )
        {
            psp_sleep(200);
            ...
        }

        d_dbrep_stop(hREP);
        d_dbrep_term(hREP);
    }
}
```

d_dbrep_connect

Begin mirroring or replicating a database

Prototype

```
int32_t d_dbrep_connect(
    const DBREP_CONNECT_PARMS *params,
    uint16_t                  *slaveId);
```

```
int32_t d_dbrep_connectW(
    const DBREP_CONNECT_PARMSW *params,    /* structure with UNICODE strings */
    uint16_t                   *slaveId);
```

Parameters

params	(input)	A pointer to a DBREP_CONNECT_PARAMS structure.
slaveId	(output)	Sequence number of the DataFlow utility's activity.

Description

This function performs the action of `dbget -b`, requesting a slave DataFlow utility (`dbmirror`, `dbrep`, `dbrepsql`) to begin a mirroring or replication connection with a master DataFlow utility. To end the connection, the corresponding function `d_dbrep_disconnect` is called.

DBREP_CONNECT_PARAMS structure definition

Element	Declaration	Description
quiet	DB_BOOLEAN	No output except for errors.
dburl	const char/wchar_t *	Fully qualified database name: `name[@tfs-domain[:port]]`
dbuserid	const char/wchar_t *	UserID to use when logging in (default: NULL).
hostname	const char/wchar_t *	The TFS domain name for the slave database. Used with `port`.
dsn	const char/wchar_t *	dbrepsql data source name: dsn;user;pswd
synchronize	int16_t	1 to begin persistent synchronous replication, -1 to end synchronous replication.
port	uint16_t	TFS anchor port. Should match TFS `-p port` number of slave (use 21553 if default). Used with `hostname`.
sql_slave_type	REP_SLAVE_TYPE	For `dbrepsql` only, the target slave server (`RST_RDMS, RST_ORACLE, RST_MSSQL, RST_MYSQL`).

Element	Declaration	Description
override_inmem	uint16_t	Force slave database to disk if master database is in-memory.
slave_done_notifier	SLAVE_DONE_FCN	Callback function. Called when the slave thread ends. See below.
stdout_file	const char/wchar_t *	This specifies the name of the file to which output data will be written. If set to PSP_STDOUT, stdout is used. If set to NULL or to an empty string, no output will be generated.

SLAVE_DONE_FCN Prototype

```
void my_slave_done_fcn(
    int32_t    slave_return)
```

SLAVE_DONE_FCN Parameters

slave_return	(input)	Return code from slave thread.

SLAVE_DONE_FCN Description

When slave_done_notifier is not NULL, it is a callback function. The slave is started in dbmirror/dbrep/dbrepsql in a thread. When the slave thread exits, either by success or failure, the function specified in slave_done_notifier is called with the return code from the slave thread, S_OKAY or some error code. The slave_done_notifier only needs to be specified if an application needs to know when a slave ends. If you specify a callback function in slave_done_notifier, the value returned in slaveId will always be 0.

Reference Manual

Database Replication Utility

Database Get Utility

Required Headers

```
#include "rdm.h"
#include "mirutils.h"
```

Libraries

Library Name	Description
rdmerdm10	RDMe Runtime Library
rdmedbmirror10	Mirroring/Replication Control Functions Library

See Library Naming Conventions section for full library name and a list of library dependencies.

Required Packages

High Availability, or

Flow

Return Codes

Value	Name	Description
0	S_OKAY	A normal return; everything worked successfully
3	S_DUPLICATE	An attempt was made to store a duplicate value into a unique key field.
-904	S_NOMEMORY	The system is unable to allocate sufficient memory.
-924	S_INVNULL	Invalid NULL parameter.

See Also

d_dbrep_disconnect, d_dbrep_start , d_dbrep_stop, d_dbrep_term

d_dbrep_disconnect

Stop mirroring or replicating a database

Prototype

```
int32_t d_dbrep_disconnect(
    const DBREP_DISCONNECT_PARMS *params);
```

```
int32_t d_dbrep_disconnectW(
    const DBREP_DISCONNECT_PARMSW *params);   /* structure with UNICODE string */
```

Parameters

`params`	`(input)`	A pointer to a DBREP_DISCONNECT_PARAMS structure.

Description

This function performs the action of `dbget -e`, requesting a slave data flow utility (`dbmirror`, `dbrep`, `dbrepsql`) to terminate a mirroring or replication connection with a master data flow utility. To initiate the connection, the corresponding function d_dbrep_connect is called.

DBREP_CONNECT_PARAMS structure definition

Element	Declaration	Description
`quiet`	`DB_BOOLEAN`	No output except for errors.
`dburl`	`const char/wchar_t *`	Fully qualified database name: name[@tfsdomain[:port]]
`dbuserid`	`const char/wchar_t *`	UserID to use when logging in (default: NULL).
`hostname`	`const char/wchar_t *`	The TFS domain name for the slave database. Used with `port`.
`port`	`uint16_t`	TFS anchor port. Should match TFS `-p port` number of slave (use 21553 if default). Used with `hostname`.
`stdout_file`	`const char/wchar_t *`	This specifies the name of the file to which output data will be written. If set to PSP_STDOUT, stdout is used. If set to NULL or to an empty string, no output will be generated.

Reference Manual

Database Replication Utility

Database Get Utility

Required Headers

```
#include "rdm.h"
#include "mirutils.h"
```

Libraries

Library Name	Description
rdmerdm10	RDMe Runtime Library
rdmedbmirror10	Mirroring/Replication Control Functions Library

See Library Naming Conventions section for full library name and a list of library dependencies.

Required Packages

High Availability , or

Flow

Return Codes

Value	Name	Description
0	S_OKAY	A normal return; everything worked successfully
2	S_NOTFOUND	A record was not found. The key find function returns S_NOT-FOUND when the requested key is not found. The first, last, next, and previous functions return this value when there are no more records to find. The next and previous functions will wrap to the beginning and end of the files, respectively, after this status is returned. For BLOB functions, the S_NOTFOUND return code means no BLOB exists (either it was never set with d_blobwrite, or the entire thing has been deleted with d_blobdelete or with d_blobtruncate at position 0).
-214	S_TX_CONNECT	Failed to connect to the TFS

See Also

d_dbrep_connect, d_dbrep_start , d_dbrep_stop, d_dbrep_term

d_dbrep_init

Provide the replication utility with parameters to start up as a server

Prototype

```
int32_t d_dbrep_init(
    const DBREP_INIT_PARAMS *rparams,
    REP_HANDLE              *hREP);
```

```
int32_t d_dbrep_initW(
    const DBREP_INIT_PARAMSW *rparamsW,  /* structure with UNICODE string */
    REP_HANDLE               *hREP);
```

Parameters

rparams	(input)	Parameters required for dbrep operation.
hREP	(output)	Handle used for running, stopping or terminating this server.

Description

This function initializes the functionality of the replication utility so that it can operate as a server, and provides a handle to control the utility.

When d_dbrep_init has been called, then d_dbrep_term should be called prior to termination of the application program.

Once this function has been called, the mirroring utility can be controlled through the functions d_dbrep_start, d_dbrep_stop, and d_dbrep_term.

DBREP_INIT_PARAMS structure definition

Element	Declaration	Description
docroot	char/wchar_t	Document root - should match that of the corresponding TFS.
port	uint16_t	TFS anchor port. Should match TFS -p port number (use 21553 if default).
verbose	uint16_t	Verbose, true or false.
diskless	uint16_t	Diskless, true or false.
stdout_file	const char *	File name if stdout is to be redirected to a file, else set to PSP_STDOUT

Reference Manual

Database Replication Utility

Required Headers

```
#include "rdm.h"
#include "mirutils.h"
```

Libraries

Library Name	Description
rdmerdm10	RDMe Runtime Library
rdmedbmirror10	Mirroring/Replication Control Functions Library
rdmedbrep10	Replication to RDM Embedded Library

See Library Naming Conventions section for full library name and a list of library dependencies.

Required Packages

Flow

Return Codes

Value	Name	Description
0	S_OKAY	A normal return; everything worked successfully
-904	S_NOMEMORY	The system is unable to allocate sufficient memory.
-924	S_INVNULL	Invalid NULL parameter.

See Also

d_dbmir_init, d_dbrepsql_init, d_dbrep_start, d_dbrep_stop, d_dbrep_term

Example

```
int main(int argc, char *argv[])
{
    DBREP_INIT_PARAMS  rparams;
    REP_HANDLE         hREP;
    uint16_t           rep_done = 0;
    int32_t            rc;

    memset(&rparams, 0, sizeof(DBREP_PARAMS));

    rparams.docroot     = "c:\\RDMe\\databases";
    rparams.port        = 21553;
    rparams.stdout_file = PSP_STDOUT;
```

```
    rc = d_dbrep_init(&rparams, &hREP);

    if (rc == S_OKAY)
    {
        /* Starts mirroring server code to process requests */
        d_dbrep_start(hREP, TRUE, &rep_done);

        /* wait for termination conditions */
        while (!rep_done)
        {
            psp_sleep(200);
            ...
        }

        d_dbrep_stop(hREP);

        d_dbrep_term(hREP);
    }
}
```

d_dbrep_start

Begin execution of an initialized data flow server utility

Prototype

```
int32_t d_dbrep_start(
    REP_HANDLE  hREP,
    DB_BOOLEAN  threaded
    uint16_t    rep_done);
```

Parameters

hREP	(input)	Handle returned from a successful data flow initialization call.
threaded	(input)	If TRUE, start a new thread for processing utility functionality.
rep_done	(input)	Set to TRUE when data flow thread has terminated and cleaned up.

Description

This function begins data flow server utility operation within the program space of the caller. The handle, hREP, must have been obtained through a data flow initialization call, which must have been successful.

The control functions, **d_dbrep_start**, **d_dbrep_stop**, and **d_dbrep_term** all operate on a hREP handle that has been initialized as a **dbrep**, **dbrepsql**, or **dbmirror** utility through the corresponding initialization function, **d_dbrep_init**, **d_dbrepsql_init**, or **d_dbmir_init**.

If threaded is TRUE, this function will return immediately. This allows the calling program to proceed with other calls to the RDM Embedded core functions, or not. To terminate the server cleanly at a later time, the **d_dbrep_stop** function may be used. When threaded is FALSE, the calling program will not return from this function unless there is an error. To terminate the utility when threaded is FALSE, the program must be externally terminated.

> Note that this does not begin mirroring or replication. This only begins the server that will handle mirroring or replication requests. See **d_dbrep_connect** to begin the mirroring process.

Reference Manual

Database Replication Utility

Required Headers

```
#include "rdm.h"
#include "mirutils.h"
```

Libraries

Library Name	Description
`rdmerdm10`	RDMe Runtime Library
`rdmedbmirror10`	Mirroring/Replication Control Functions Library

See Library Naming Conventions section for full library name and a list of library dependencies.

Return Codes

Value	Name	Description
0	S_OKAY	A normal return; everything worked successfully
-43	S_INVREPHANDLE	Invalid replication handle has been provided.
-200	S_TX_ERROR	Generic TFS Error
-216	S_TX_LISTEN	TCP/IP listen function failed.
-904	S_NOMEMORY	The system is unable to allocate sufficient memory.
-924	S_INVNULL	Invalid NULL parameter.

Required Packages

High Availability , or

Flow

See Also

d_dbmir_init, d_dbrep_init, d_dbrepsql_init , d_dbrep_stop, d_dbrep_connect, d_dbrep_disconnect, d_dbrep_term

d_dbrep_stop

Terminate data flow server utility

Prototype

```
int32_t d_dbrep_stop(
    REP_HANDLE hREP);
```

Parameters

hREP	(input)	Handle returned from a successful data flow initialization call.

Description

This function terminates the data flow server utility running within the program space of the caller. The handle, hREP, must have been obtained through a **d_dbrep_init**, **d_dbrepsql_init**, or **d_dbmir_init** call, which must have been successful.

This function may be called if **d_dbrep_start** has been called with threaded == TRUE. It performs a clean shut-down of the threads that are servicing external database programs.

Reference Manual

Database Replication Utility

Required Headers

```
#include "rdm.h"
#include "mirutils.h"
```

Libraries

Library Name	Description
rdmerdm10	RDMe Runtime Library
rdmedbmirror10	Mirroring/Replication Control Functions Library

See Library Naming Conventions section for full library name and a list of library dependencies.

Return Codes

Value	Name	Description
0	S_OKAY	A normal return; everything worked successfully

-43	S_INVREPHANDLE	Invalid replication handle has been provided.

Required Packages

High Availability , or

Flow

See Also

d_dbmir_init, d_dbrep_init, d_dbrepsql_init , d_dbrep_start, d_dbrep_term

d_dbrep_term

Clean up resources allocated by data flow utility initialization

Prototype

```
int32_t d_dbrep_term(
    REP_HANDLE hREP);
```

Parameters

`hREP`	`(input)`	Handle returned from a successful data flow initialization call.

Description

This function frees resources allocated by the data flow utility during and after the call to the initialization function. The utility thread must be stopped if it has been started, prior to calling this function.

Reference Manual

Database Replication Utility

Required Headers

```
#include "rdm.h"
#include "mirutils.h"
```

Libraries

Library Name	Description
`rdmerdm10`	RDMe Runtime Library
`rdmedbmirror10`	Mirroring/Replication Control Functions Library

See Library Naming Conventions section for full library name and a list of library dependencies.

Return Codes

Value	Name	Description
0	S_OKAY	A normal return; everything worked successfully
-43	S_INVREPHANDLE	Invalid replication handle has been provided.

Required Packages

High Availability , or

Flow

See Also

d_dbmir_init, d_dbrep_init, d_dbrepsql_init, d_dbrep_start, d_dbrep_stop

d_dbrepsql_init

Provide the SQL replication utility with parameters to start up as a server

Prototype

```
int32_t d_dbrepsql_init(
    const DBREP_INIT_PARAMS *rparams,
    REP_HANDLE              *hREP);
```

```
int32_t d_dbrepsql_initW(
    const DBREP_INIT_PARAMSW *rparamsW,  /* structure with UNICODE string */
    REP_HANDLE               *hREP);
```

Parameters

rparams	(input)	Parameters required for dbrepsql operation.
hREP	(output)	Handle used for running, stopping or terminating this server.

Description

This function initializes the functionality of the SQL replication utility so that it can operate as a server, and provides a handle to control the utility.

When d_dbrep_init has been called, then d_dbrep_term should be called prior to termination of the application program.

Once this function has been called, the mirroring utility can be controlled through the functions d_dbrep_start, d_dbrep_stop, and d_dbrep_term.

DBREP_INIT_PARAMS structure definition

Element	Declaration	Description
docroot	char/wchar_t	Document root - should match that of the corresponding TFS.
port	uint16_t	TFS anchor port. Should match TFS -p port number (use 21553 if default).
verbose	uint16_t	Verbose, true or false.
diskless	uint16_t	Diskless, true or false.
stdout_file	const char *	File name if stdout is to be redirected to a file, else set to PSP_STDOUT

Reference Manual

Database Replication Utility

Required Headers

```
#include "rdm.h"
#include "mirutils.h"
```

Libraries

Library Name	Description
rdmerdm10	RDMe Runtime Library
rdmedbmirror10	Mirroring/Replication Control Functions Library
rdmedbrepsql10	Replication to SQL Library

See Library Naming Conventions section for full library name and a list of library dependencies.

Required Packages

Flow

Return Codes

Value	Name	Description
0	S_OKAY	A normal return; everything worked successfully
-904	S_NOMEMORY	The system is unable to allocate sufficient memory.
-924	S_INVNULL	Invalid NULL parameter.

See Also

d_dbmir_init, d_dbrep_init, d_dbrep_start, d_dbrep_stop, d_dbrep_term

Example

```
int main(int argc, char *argv[])
{
    DBREP_INIT_PARAMS  rparams;
    REP_HANDLE         hREP;
    uint16_t           rep_done = 0;
    int32_t            rc;

    memset(&rparams, 0, sizeof(DBREP_PARAMS));

    rparams.docroot     = "c:\\RDMe\\databases";
    rparams.port        = 21553;
    rparams.stdout_file = PSP_STDOUT;
```

```
    rc = d_dbrepsql_init(&rparams, &hREP);

    if (rc == S_OKAY)
    {
        /* Starts mirroring server code to process requests */
        d_dbrep_start(hREP, TRUE, &rep_done);

        /* wait for termination conditions */
        while (!rep_done)
        {
            psp_sleep(200);
            ...
        }

        d_dbrep_stop(hREP);

        d_dbrep_term(hREP);
    }
}
```

DataFlow and Mirroring Utilities

dbmirror

Database Mirroring Server

Prototype

```
dbmirror [-d PATH] [-p PORT] [-v] [-nodisk]
          [-start | -stop | -query | -install exepath | -uninstall]
```

Description

The **dbmirror** utility acts as either a *master* or *slave* database mirroring server. The same **dbmirror** instance may operate in both master and slave roles at the same time (using separate threads), and may operate on any number of databases, but each database is either a master or a slave database. It always runs on the same computer as a TFS (the tfserver utility or a customized TFS created by the programmer). The TFS/**dbmirror** partnership is established by using successive TCP/IP ports (e.g. if the TFS uses port 2335, then dbmirror will use 2336).

In the master role, **dbmirror** responds to requests from counterpart dbmirror slaves. For the initial request from the dbmirror slave, the dbmirror master will create a thread to service the database being requested. The thread will be dedicated to serving log files to the slave until the slave terminates the connection. The master thread will receive log file requests from the slave, and search the local directory for each request. If the log file exists, it will send it to the slave. If it doesn't exist, and the master hasn't already done so, it will send the entire database to the slave, after which the slave will begin requesting each log file that has been created after the database image was generated.

In the slave role, **dbmirror** may partner with a TFS or not. If there is a TFS, the changes are applied to the local database. If there is no TFS, then **dbmirror** will save change log files in the database's subdirectory (below the document root), which may be processed at a later time by a TFS. It is the slave that initiates the connection with the master dbmirror. The slave initiates the connection when the **dbget** utility is run with command-line options that identify this **dbmirror** and a database that is hosted by another TFS. In response to dbget, **dbmirror** will begin a slave thread which connects to the master **dbmirror** and requests updates from the database identified in the dbget command-line.

By convention, the slave database mirror is placed within a directory that is named after the host of the master database. For example, if the database named `tims` is mirrored from a host identified as `tfs.raima.com`, then the mirror of `tims` will be stored in:

```
documentRoot/tfs.raima.com-master_port/tims
```

All databases mirrored to this document root from the same host will be stored in the same subdirectory. Databases stored in subdirectories are the products of mirroring or replication, and thus cannot be opened for updating. See Opening Slave Databases from Programs for more information about using read-only databases from programs.

When this utility is run as a foreground process, it may be cleanly terminated by issuing SIGINT (normally ^C from the keyboard). SIGTERM has the same effect. If the utility does not terminate immediately it will be terminated immediately (cleanly or not) upon the third signal.

Options

`-d PATH`	The path of the *Document Root*. Default is current directory.
`-p PORT`	Anchor PORT, referring to the TFS associated with this utility. Default is 21553.
`-v`	Verbose output.
`-nodisk`	Don't use any disk I/O.
`-start`	Begin execution of utility as a separate process.
`-stop`	Stop execution of utility as a separate process.
`-query`	Check the status of the utility process (running/not running).
`-install PATH`	**Windows only,** Requires administrative privilege. Install utility as an automatic service. *PATH* is the directory or fully qualified path to the utility's executable file.
`-uninstall`	**Windows only,** Requires administrative privilege. Uninstall the utility service. Use the service control manager to stop the execution of the service first.

DataFlow User's Guide

Data Flow Architecture

Database Storage Location

Required Packages

High Availability, or

DataFlow

dbrep, dbrepsql

Database Replication Slave Utility

Prototype

```
dbrep{sql} [-d PATH] [-p PORT] [-v] [nodisk]
            [-start | -stop | -query | -install PATH | -uninstall]
```

Description

The **dbrep** utility is a data flow utility that performs replication only. **dbrep**may be in master or slave roles. When in the master role, it can serve files to **dbmirror**, **dbrep** or **dbrepsql** slaves. The same **dbrep** utility may act as both master and slave at the same time. The same **dbrep** slave may operate on any number of databases from any number of different master database locations. It always runs on the same computer as a TFS (the tfserver utility or a customized TFS created by the programmer). The TFS/**dbrep** partnership is established by using successive TCP/IP ports (e.g. if the TFS uses port 2335, then dbrep is using 2336).

Dbrep requires a TFS to be running on the same computer. The **dbrep**utility initiates a connection to the master dbmirror when the **dbget** utility is run with command-line options that identify this **dbrep** and a database that is hosted by another TFS. In response to **dbget**, **dbrep** will begin a slave thread which connects to the master dbmirror and requests updates from the database identified in the dbget command-line.

The **dbrepsql** utility operates in the same manner as **dbrep**, except that it replicates to SQL databases. **Dbrep** creates and maintains an RDM Embedded database, but **dbrepsql** maintains a SQL database that has been externally created and made available through ODBC.

When **dbget** notifies **dbrepsql** to begin replication of a database, it will have supplied the destination type (that is, RDM Server, Oracle, Microsoft SQL Server, or MySQL), because there are slight differences in the sequence of ODBC function calls required for each one. When the **dbget** command-line doesn't identify one of the SQL database types, it defaults to RDM Server.

To locate the ODBC data source, **dbrepsql** reads the environment variable named `RDSLOGIN`. In an RDM Server environment, which variable is already used to locate the server. When using other types of SQL database servers, the same environment variable will be used to identify the ODBC data source, the username, and password, as follows:

```
RDSLOGIN=Oracle-svr;WLWadmin;Jn@@f0
```

Alternately, the data source can be specified as the -dsn option to the dbget command.

This database must exist before **dbrepsql** is asked to add its updates from the master. The SQL DDL used to define the database must have been derived from the original RDM Embedded DDL file, and the schemaxlate utility must have been used to create a customized SQL DDL file.

By convention, the slave database replicate created by **dbrep** is placed within a directory that is named after the host of the master database. For example, if the database named `tims` is mirrored from a host identified as `tfs.raima.com`, then the replicate of `tims` will be stored in:

```
documentRoot/tfs.raima.com-master_port/tims
```

All databases replicated to this document root from the same host will be stored in the same subdirectory. Databases stored in subdirectories are the products of mirroring or replication, and thus cannot be opened for updating. See Opening Slave Databases from Programs for more information about using read-only databases from programs.

When this utility is run as a foreground process, it may be cleanly terminated by issuing SIGINT (normally ^C from the keyboard). SIGTERM has the same effect. If the utility does not terminate immediately it will be terminated immediately (cleanly or not) upon the third signal.

Options

`-d PATH`	The path of the *Document Root*. Default is current directory.
`-p PORT`	Anchor PORT, referring to the TFS associated with this utility. Default is 21553.
`-v`	Verbose output.
`-nodisk`	Don't use any disk I/O.
`-start`	Begin execution of utility as a separate process.
`-stop`	Stop execution of utility as a separate process.
`-query`	Check the status of the utility process (running/not running).
`-install PATH`	**Windows only,** Requires administrative privilege. Install utility as an automatic service. *PATH* is the directory or fully qualified path to the utility's executable file.
`-uninstall`	**Windows only,** Requires administrative privilege. Uninstall the utility service. Use the service control manager to stop the execution of the service first.

DataFlow User's Guide

Data Flow Architecture

Database Storage Location

Required Packages

DataFlow

dbget

Database Mirroring Server

Prototype

```
dbget [-s SLAVE_HOSTNAME] [-p SLAVE_ANCHOR_PORT] [-sync] [-unsync] [-b] [-e] [-
override_inmem]
       [-dsn dsn;user;pswd] [-oracle|-mssql|-mysql] dbname[@master_tfs_
domain][:master_tfs_port]
```

Description

The **dbget** utility notifies a *data flow slave utility* that it should begin mirroring or replicating a database. A data flow slave utility is one of the following:

- dbmirror - the database will be *mirrored*.
- dbrep - the database will be *replicated* to a slave RDM Embedded database.
- dbrepsql - the database will be *replicated* to a slave ODBC data source.

Regardless of which data flow utility is running, it is identified through the `-s HOSTNAME` and `-p PORT`.

Dbget is used to obtain the original copy of the database, which is then continuously updated via mirroring or replication processes.

When the `-b` option is used, mirroring/replication will be started if it is not already active. To discontinue mirroring/replication, use **dbget** with `-e`. At least one of `-b` or `-e` must be specified.

Synchronous mirroring is started by including the `-sync`. This option is invalid for replication (dbrep or dbrepsql). Synchronous mirroring is persistent, meaning that if the mirroring client stops or is disconnected, the mirroring will remain synchronous the next time **dbget** is used. Beware that a synchronous mirror slave can block transactions applied to the master if the slave is not active.

When dbrepsql is the data flow utility being notified, it is necessary to identify the type of database. RDM Server is the default, and no option is needed. For Oracle, MySQL, or Microsoft SQL Server, use options `-oracle`, `-mysql`, or `-mssql`, respectively.

Options

`-s HOSTNAME`	HOSTNAME of slave {dbmirror, dbrep or dbrepsql} (default is `localhost`).
`-p PORT`	TCP/IP anchor port of slave {dbmirror, dbrep or dbrepsql} (default is `21553`).
`-sync`	Valid only when notifying dbmirror. Mirrored database is synchronous.
`-unsync`	Valid only when notifying dbmirror. End persistent synchronous mirroring.
`-b`	Begin mirroring or replication.
`-e`	End mirroring or replication.
`-override_inmem`	If the master database is in-memory, force slave to be written to disk.
`-dsn dsn:user:pswd`	Specify DSN for ODBC connection to slave SQL server (dbrepsql only).
`-oracle`	Slave dbrepsql connects to Oracle server.
`-mssql`	Slave dbrepsql connects to Microsoft SQL server.
`-mysql`	Slave dbrepsql connects to MySQL server.
`dbname[@tfs_domain][:tfs_port]`	Name and location of master database server. Default domain is `localhost`. Default port is `21553`. An incorrect port number will return a error S_NOMIR-SERVER,

DataFlow User's Guide

Data Flow Utilities

Required Packages

High Availability, or

DataFlow

schemaxlate

Convert an RDM Embedded database dictionary into a compatible SQL schema.

Prototype

```
schemaxlate [-h] [-B] [-V] [-n] [-f] [-o outfile] [-d device]
            [-oracle|-mssql|mysql] dbname
```

Description

Use the Schema Translate utility to create a SQL schema from an RDM Embedded database dictionary. By default, the output file will be compatible with RDM Server SQL. Optionally, the output file will be made compatible with Oracle, Microsoft SQL Server, or MySQL.

The *dbname* is a database that already exists in the RDM Embedded environment, having already been compiled from a DDL file and ddlp. This utility will "reverse engineer" the compiled DDL to produce the SQL file. The SQL file can be compiled by the DDL processor of the target SQL product.

Use this tool to generate SQL DDL for creating/expanding a SQL database to be a recipient of replicated data from RDM Embedded. See Database Mirroring and Replication for more information.

Options

`-h`	Display this usage information
`-B`	Do not display the banner
`-V`	Display the version information
`-n`	Force fields to allow NULLs. By default, "NOT NULL" will be added to all fields of a "core" database.
`-f`	Force overwrite of existing output file
`-o` *`filespec`*	Name of output file, default is *dbname*_rdms.sql. Also overrides the *-oracle*, *-mssql* or *-mysql* default out file names.
`-d` *`devname`*	Name of "device" for files, default is **sqldev**. This is an RDM Server device that specifies the location where the database will be created. "Device" is a name associated to a directory set by the RDM Server administration tool.
`-oracle`	Create out compatible with Oracle (default output file is *dbname*_oracle.sql).
`-mssql`	Create output compatible with Microsoft SQL Server (default output file is *dbname*_mssql.sql).
`-mysql`	Create output compatible with MySQL (default output file is *dbname*_mysql.sql).