Note

Before using this information and the product it supports, read the information in "Notices" on page 491.
Contents

About this book .................................................. ix
Assumptions about your locale ................................ ix
How to send your comments .................................... ix

Part 1. Overview of DB2 Data Stream Engine .......... 1

Chapter 1. DSE architecture ............................... 3
Chapter 2. DSE terminology ............................... 5

Part 2. Installing DB2 Data Stream Engine ............. 7

Chapter 3. System requirements for DSE ............... 9
Software requirements (DSE) .................................. 9
Hardware requirements (DSE) .................................. 10

Chapter 4. Installing DSE .................................... 11
Chapter 5. DSE directory structure ........................ 13

Chapter 6. Tutorial: Getting started with DSE ........ 17
Setting up a test storage system ........................... 17
Preparing the DB2 instance ................................. 17
Setting up the environment variables ..................... 18
Verifying your installation with the quick start script 19
Verifying your installation step by step ................... 19
Setting up a store ............................................. 20
Building a simple data feed simulator and feed handler ................................. 21
Configuring DSE to verify the installation ............... 22
Starting DSE to verify the installation .................... 23
Setting up a wrapper to query a DB2 database ........... 23

Part 3. Administering DSE ................................. 27

Chapter 7. rtloader command: Starting DSE ........... 29

Chapter 8. Utility programs ............................... 31
rtldiscarddmp utility ......................................... 31
rtllmode utility .............................................. 32
rtlsnmp utility .............................................. 40
rtlsstat utility .............................................. 43
rtl_make_pass utility ........................................ 49

Chapter 9. System configuration parameters .......... 51
General system configuration parameters ............... 52
ENABLE_NEWSYM .......................................... 52
INCLUDE .................................................... 52
NODISKWRITE ............................................. 52
RESTART_ENABLE ......................................... 52
RTL_LOG_LOCALTIME ..................................... 53
RTL_LOG_PATH ............................................. 53
RTL_SYS_MSG_PATH ....................................... 53
UDPX ....................................................... 54

Feed configuration parameters ............................ 54
FEEDXX_ACCEPT_TICKS_ALWAYS ......................... 54
FEEDXX_GAPFILL ........................................... 54
FEEDXX_HANDLER_NAME ................................... 54
FEEDXX_INFO ............................................. 55
FEEDXX_TRANSPORT_NAME ................................. 55
FEEDXX_SOURCE ........................................... 55
FEEDXX_STORE_NAME ..................................... 55
FEEDXX_TARGET ........................................... 55
FEEDXX_WRITECANBLOCK .................................. 56

Feed handler configuration parameters .................. 56
HANDLERTXX_INFO ......................................... 56
HANDLERTXX_LIB_PATH ................................... 56
HANDLERTXX_MSG_PATH .................................. 56
HANDLERTXX_NAME ........................................ 57

Group configuration parameters ........................... 57
GROUPXX_AUTHLEVEL ...................................... 57
GROUPXX_NAME ............................................. 57
GROUPXX_PASSWORD ....................................... 57

High availability configuration parameters ............. 57
HA_COMM_TIMEOUT ........................................ 58
HA_ENABLE .................................................. 58
HA_INFO .................................................... 58
HA_LIB_PATH ............................................... 58
HA_MAXMSG_SIZE ......................................... 58
HA_MSG_PATH ............................................... 59
HA_RETRY_DELAY ........................................... 59
HA_SAVEBUF_SIZE ........................................ 59
HA_SHARED_DB ............................................. 59
HA_STANDBY_API .......................................... 59
HA_STANDBY_PUB .......................................... 60
HA_SYNC_DELAY ........................................... 60
HAXXX_PEER_INFO .......................................... 60
HAXXX_PEER_NAME ......................................... 60

Query server configuration parameters .................. 60
MAX_QUERY_CONNECTS ....................................... 61
QUERY_THREADS ........................................... 61
RTL_SERVICE_NAME ........................................ 61
Record configuration parameters ........................ 61
RECORDXX_NAME .......................................... 61
RECORDXX_META_SCHEMA ................................ 61
RECORDXX_META_SEQUENCE ............................... 64
RECORDXX_SCHEMA ........................................ 65
RECORDXX_SEQUENCE ...................................... 67

Shared memory configuration parameters ................ 67
ADD_SHMSIZE ............................................. 67
FLBS ....................................................... 68
| Chapter 19. Enabling error recovery for storage systems | 131 |
| Chapter 20. Correcting and reloading bad data | 133 |
| Creating a discard table for bad data | 133 |
| Chapter 21. SQL queries with the DSE wrapper | 137 |
| Supported query predicates | 137 |
| Multiple predicates | 138 |
| DSE wrapper error codes | 139 |
| Chapter 22. Wrapper objects | 141 |
| Wrapper | 141 |
| Server | 142 |
| User mapping | 143 |
| Nickname | 143 |
| rtlElem UDF | 148 |
| Part 5. Developing DSE components and applications | 151 |
| Chapter 23. Feed handlers | 153 |
| Approaches to handling data formats | 154 |
| DSE record structures | 154 |
| Feed data processing | 155 |
| Tick storage | 155 |
| Metadata updates | 155 |
| Message publishing | 155 |
| Feed handler reference and context information | 156 |
| Control flows for feed handlers | 156 |
| Feed handler initialization | 157 |
| Feed initialization | 157 |
| Per message processing | 157 |
| Feed shutdown | 158 |
| Feed handler shutdown | 158 |
| Error handling and logging for feed handlers | 158 |
| Trace mechanism for feed handlers | 159 |
| Building and deployment of feed handlers | 161 |
| Examples of feed handlers | 161 |
| Chapter 24. Transports | 163 |
| Transport properties | 164 |
| Transport control flows | 164 |
| Transport handler initialization | 165 |
| Transport initialization | 165 |
| Data source and target switching | 166 |
| Feed pause and resume | 167 |
| Per message processing | 168 |
| Transport shutdown | 169 |
| Transport handler shutdown | 169 |
| Error handling and logging for transport handlers | 170 |
| Trace mechanism for transport handlers | 170 |
| Development and deployment of transport handlers | 171 |
| Supplied transport handlers | 171 |
| Source and target names for pre-built transports | 171 |
| TRANSPORTXX_INFO parameter for pre-built transports | 172 |
| Restrictions for pre-built transports | 172 |
| Chapter 25. DSE interfaces | 175 |
| Feed handler interface | 175 |
| rtl_handler_init() | 176 |
| rtl_handler_shutdown() | 177 |
| rtl_feed_init() | 178 |
| rtl_feed_shutdown() | 179 |
| rtl_feed_get_tick_from_message() | 180 |
| rtl_feed_update_metadata() | 182 |
| rtl_feed_get_publish_message() | 184 |
| rtl_feed_get_message_fmts() | 186 |
| Generic storage support interfaces | 187 |
| Storage system interface | 187 |
| Store interface | 193 |
| Partitioning interface | 197 |
| Symbol adding interface | 203 |
| Symbol scanning interface | 212 |
| Symbol updating interface | 217 |
| Tick flushing interface | 224 |
| Tick scanning interface | 235 |
| High availability communication library interface | 243 |
| rhl_ha_init() | 244 |
| rhl_hacomm_connect() | 245 |
| rhl_hacomm_disconnect() | 246 |
| rhl_hacomm_listen() | 247 |
| rhl_hacomm_peer_id() | 248 |
| rhl_hacomm_receive() | 249 |
| rhl_hacomm_send() | 250 |
| rhl_ha_shutdown() | 251 |
| Transport interface | 252 |
| rtl_tpt_msgfmt_t data structure | 253 |
| rtl_transport_get_property() | 256 |
| rtl_transport_handler_init() | 258 |
| rtl_transport_handler_shutdown() | 259 |
| rtl_transport_init() | 260 |
| rtl_transport_shutdown() | 261 |
| rtl_transport_source_batch_end() | 262 |
| rtl_transport_source_close() | 264 |
| rtl_transport_source_delete_message() | 265 |
| rtl_transport_source_get_message() | 266 |
| rtl_transport_source_open() | 268 |
| rtl_transport_source_put_message() | 269 |
| rtl_transport_source_register_callback() | 270 |
| rtl_transport_target_close() | 271 |
| rtl_transport_target_open() | 272 |
| rtl_transport_target_put_message() | 273 |
| Chapter 26. Writing client applications | 275 |
| Connecting C applications to DSE | 278 |
| Connection information | 278 |
| Connection authorization | 279 |
Connection failure due to version incompatibility
Reconnecting C applications
Querying data in DSE data stores
Querying a range of data
Querying a single element at a point in time
Querying metadata
Interpreting result records returned by DSE
Updating symbol metadata in a data store.
Querying statistics from DSE
Disconnecting C applications from DSE

Chapter 27. DSE API reference

Types of API functions
  C client API
  Feed handler API
  High availability API
  Java client API
  Record type API
  Store API
  Storage system API
  Time API
  Transport API
  DSE API functions

rtl_add_seconds()
rtl_attach()
rtl_begin_scan()
rtl_client_log()
rtl_client_log_done()
rtl_client_log_init()
rtl_detach()
rtl_elem()
rtl_end_scan()
rtl_feed_get_gapfill()
rtl_feed_get_handler_id()
rtl_feed_get_handler_name()
rtl_feed_get_info()
rtl_feed_get_source()
rtl_feed_get_store_id()
rtl_feed_get_store_name()
rtl_feed_get_target()
rtl_feed_get_transport_id()
rtl_feed_get_user_data()
rtl_feed_is_configured()
rtl_feed_log()
rtl_feed_source_get_message()
rtl_feed_source_put_message()
rtl_feed_state_change_register()
rtl_get_feed_for_symbol()
rtl_get_metadata()
rtl_get_meta_info()
rtl_get_meta_rec_info()
rtl_get_next()
rtl_get_recinfo_id_from_name()
rtl_get_rtltime_date_fields()
rtl_get_rtltime_fields()
rtl_get_rtltime_fields()
rtl_get_tick_info()
rtl_get_tick_rec_info()
rtl_get_udp_value()
About this book

This information describes the IBM® DB2® Data Stream Engine (DSE) and how to use it.

This information is written for the following audience:
- System administrators who install DSE
- Database administrators who control access to databases that use DSE
- Developers who write applications using DSE

This information assumes that you have the following background:
- A working knowledge of the Unix operating system and its utilities.
- Some experience working with relational databases or exposure to database concepts.
- Some experience programming in both the C and JAVA languages.
- Some experience with database server administration, operating-system administration, or network administration.


Assumptions about your locale

DSE supports only the English language for this release.

The examples in this manual are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for date, time, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 Data Stream Engine information. You can use any of the following methods to provide comments:


  Use this feedback page to enter and send comments.
- Send your comments by e-mail to docinf@us.ibm.com. Be sure to include the name of the product, the version number of the product, and the name and part number of the book (if applicable). If you are commenting on specific text, please include the location of the text (for example, a chapter and section title, a table number, or a page number).
Part 1. Overview of DB2 Data Stream Engine

DB2 Data Stream Engine (DSE) enables organizations to store and forward high volumes of data from multiple data streams. DSE can load the data into the DB2 database system and make that data available to queries in real-time through SQL.

DSE is a data stream loading solution that is available as a separately orderable product to be used with IBM DB2 Database for Linux®, UNIX®, and Windows®, Version 9.5.

The data messages from the stream (feed) can be aggregated, filtered and enriched in real time before being stored or forwarded. One example is a financial market data stream that provides information about financial transactions, such as stock trades and quotes.

DSE can load data from different kinds of feeds, including queues, files, and sockets. For example, DSE can get data from queues that are managed by WebSphere® MQ for AIX®, Linux, and Solaris and flush it to the storage system (such as a DB2 database). If the feed source is a WebSphere MQ queue, DSE can be run in restartable mode to preserve data integrity and protect against data loss if system failure occurs.

The benefits of DSE include:
- Extremely high throughput and low latency data loading.
- Simultaneous storing and publishing of data from multiple feeds (up to 100).
- Filtering and enriching of data from feeds before storing and publishing.
- The ability to maintain metadata, such as current state, for entities that are processed from the feeds.
- Simultaneous persistence of data to multiple database servers on multiple hosts (up to 100).
- Shared memory storage of data, allowing real-time access.
- Access to both real-time and historical data through standard SQL, C-API, and Java™ API interfaces.
- Recoverability of data if system failure occurs (if DSE is running in restartable mode).
Chapter 1. DSE architecture

DSE is a multi-threaded program that processes data feeds for storing and publishing. Shared libraries and shared memory are used to enable data feeds to be processed and stored with very high throughput and low latency.

DSE uses shared libraries to handle data from various data feed sources and to persist data to various storage systems. The transport libraries read messages from a feed and write them to a data target. The feed handler libraries process data from the feed. The storage system libraries persist data to the storage system.

DSE provides transport libraries for TCP/IP sockets, files, and WebSphere MQ queues. DSE provides examples of how to write a feed handler library and an implementation of a DB2 storage system library.

DSE caches data in shared memory before persisting it to the storage system. When the data is in shared memory, it is immediately available to queries (if DSE is running in non-restartable mode), so the data is effectively real time. DSE periodically persists (flushes) data from shared memory to the storage system, for example, a DB2 database. Data is not removed from shared memory at this point, so this data can continue to be accessed by queries in real time. Queries always select data from shared memory, if it is available, rather than from the storage system.

To prevent the shared memory from becoming full, you configure a rolling window of time that controls when DSE clears space in memory. A rolling window of time consists of intervals that are defined in seconds.

Example: If you configure the interval as 10 seconds, and the number of intervals as 6, the rolling window of time is 60 seconds. DSE keeps at least 60 seconds of data in memory. Every 10 seconds, the data in the oldest 10 seconds is removed.

DSE can load and store data in:
- A master-detail relationship, with entities (symbols) and events (ticks).
- A simple schema, with unassociated events (ticks).

An example of data using a master-detail relationship is the data on the New York Stock Exchange (NYSE). The stock symbol IBM is an entity and all of the trades and quotes for that symbol are the events. In addition to storing the events, DSE can maintain metadata about the entity. For example, you could maintain the current state of trading for a stock, such as cumulative volume or the highest price. This information can be:
- Initialized from the storage system on startup.
- Updated and maintained by messages from the feeds in real-time.
- Persisted to the storage system on demand or periodically.

DSE enables you to query both real-time and historical data through standard SQL, C-API, and Java API interfaces. The C and Java client APIs are described in Chapter 27, “DSE API reference,” on page 285. The DSE wrapper is described in Chapter 22, “Wrapper objects,” on page 141. The DSE wrapper allows you to query both real-time and historical data that is managed by DSE, as if it were stored in a regular database table.
You can run multiple DSE instances on the same system if:
- You have enough shared memory.
- You configure each DSE instance to use a different shared memory key.
Chapter 2. DSE terminology

Some terms that are used throughout this information have specific meaning for DSE.

These terms include:

**Feed**  An online data source from which DSE receives data. Each feed is processed by its own feed handler.

**Flush**  Data is flushed when it is written from shared memory to the database server. The term flush rate refers to the amount of data written to the database server per second.

**Ingestion rate**  The rate at which data is received through an online data feed source. The ingestion rate can be measured on a per-feed basis, or as the sum of the rates for all feeds, in which case it is referred to as the total ingestion rate.

**Metadata**  Metadata information for a particular symbol. This metadata refers to any additional information related to a symbol including reference data, current state and context. For example, the metadata for a financial market data stock feed can contain the full company name, last trade price, maximum and minimum trade prices, and the cumulative volume of shares that were traded.

**Storage system**  A mechanism for providing long-term persistence for symbols, metadata, and ticks stored by DSE. DSE stores data in shared memory for a defined rolling window of time and uses a storage system to provide longer-term access to data. The storage system can use a relational database management system such as DB2.

**Store**  A repository for data. The store includes shared memory storage and longer-term persistence that is provided by a storage system. In a DB2 storage system, a store consists of a ticks table, and optionally a symbols table.

Stores that use a master-detail relationship contain a set of symbols, metadata, and ticks of a given record format. Stores that use a simple schema contain ticks data.

**Symbol**  An entity that you are receiving data for from the online data feed. For example, a symbol for a financial market data stock feed could be a particular security such as IBM on the New York Stock Exchange.

**Thread**  A subprocess spawned by another process, that executes in parallel, but shares state information and context, with its parent process. In DSE, feed handlers and flushers are threads spawned by the main DSE process.

**Tick**  An event. In a master-detail relationship, a tick is a record of data about an event for a symbol. For example, a tick for a financial market data stock feed could be a trade record for a particular security such as IBM on the New York Stock Exchange. In a simple schema, a tick is an unassociated event, such as a record or row.
Transport

A communication mechanism for a data feed from which DSE receives data.
Part 2. Installing DB2 Data Stream Engine

Install DSE, then run the provided simple example to verify the installation. This installation verification step involves setting up the storage system and building and running a simple feed handler. You load sample data into a DB2 database and determine whether you can query that data successfully.
Chapter 3. System requirements for DSE

To get started, meet the minimum system requirements and install, configure, and start DSE. The commands that are described in the installation topics require database administrator privileges.

To use DSE, you need the required software and hardware.

Software requirements (DSE)

Meet the software requirements for DSE before installing the product.

DSE is supported on the following platforms:
- AIX 5.3 (64-bit)
- Red Hat Enterprise Linux (RHEL) 4 (32-bit)
- Red Hat Enterprise Linux (RHEL) 4 (64-bit)
- Solaris 10 (64-bit)
- SUSE Linux Enterprise Server (SLES) 10 (64-bit)

DSE supports the use of storage systems, such as:
- DB2 Version 9.1 for Linux, UNIX, and Windows with Fix Pack 3
  DB2 Version 9.1 includes the DB2 Data Partitioning Feature (DPF).
- DB2 Version 9.5 for Linux, UNIX, and Windows
  DPF is available with the base, advanced, and enterprise editions of DB2

If you use the DSE wrapper, a license for the WebSphere Federation Server is required.

DSE supports transactional feed sources, such as queues from WebSphere MQ for
AIX, Linux, and Solaris, Version 6.0. Non-transactional feed sources such as files
and TCP/IP sockets are also supported.

DSE provides source code samples that were tested using the following compilers:

<table>
<thead>
<tr>
<th>Platform</th>
<th>C compiler</th>
<th>Java Runtime Environment (JRE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX 5.3 (64-bit)</td>
<td>xlc_r Version 8.0.0.001</td>
<td>IBM J2RE Version 1.4.2</td>
</tr>
<tr>
<td>RHEL 4 (32-bit)</td>
<td>gcc Version 3.4.6</td>
<td>IBM J2RE Version 1.4.2</td>
</tr>
<tr>
<td>RHEL 4 (64-bit)</td>
<td>gcc Version 3.4.4</td>
<td>IBM J2RE Version 1.4.2</td>
</tr>
<tr>
<td>Solaris 10 (64-bit)</td>
<td>Sun WorkShop 6 cc Version 5.3</td>
<td>Sun J2RE Version 1.5.0</td>
</tr>
</tbody>
</table>
Hardware requirements (DSE)

Meet the hardware requirements for DSE before installing the product.

DSE requires the following minimum hardware:
- 2 CPUs
- 256 MB memory
- 0.5 GB hard disk
- 1.0 GHz CPU speed
Chapter 4. Installing DSE

You can install some or all of the DSE components with the installation program.

**Requirement:** If you are installing DSE and verifying the installation on a single machine, install the wrapper and the server on that machine.

To install DSE from the CD:

1. Run the following command:
   <pre>install_dse</pre>

   **Requirement:** If you want to install DSE in the default directory, complete one of the following steps:
   - Run the `install_dse` command as root.
   - Change the permissions of the default directory so that a non-root user can install DSE in that directory.

2. Read the product description page. Enter y or n at the prompt to indicate whether you want to continue installing DSE.

3. Read and scroll through the pages of the license agreement. Enter y or n at the prompt to indicate whether you accept the license agreement.

4. Specify the directory in which you want to install DSE. You can use the default directory or specify a different directory.
   - On Linux, the default directory is `/opt/ibm/dse/version`.
   - On Solaris and AIX, the default directory is `/opt/IBM/dse/version`.

   The `version` value is the DSE version number. **Requirement:** The directory in which you install DSE must be a writable directory.

5. Enter 1, 2, 3, or 4 to indicate the platform on which you want to install DSE.
   - Solaris 64-bit
   - AIX 64-bit
   - Linux 32-bit
   - Linux 64-bit

6. Install one or more of the DSE components. DSE has four components. To install one or more individual components:
   a. Enter 1, 2, 3, or 4.
   b. Enter the numbers of additional components that you want to install, or enter q to exit the installation program.

   To install all of the components:
   a. Enter 5.
   b. Enter q to exit the installation program.

The installation options are:

1. **C-API**
   Install this component to enable an application program that is written in C to communicate with the DSE server.
2. Java-API
   Install this component to enable an application program that is written in Java to communicate with the DSE server.

3. Wrapper
   Install this component to enable SQL queries that are issued in a DB2 database to get data from DSE. To complete the installation of the wrapper, be sure to complete the steps in “Preparing the DB2 instance” on page 17.

4. Server
   Install this component to use DSE.

5. All components
   Install the C-API, Java-API, wrapper, and server components of DSE.
Chapter 5. DSE directory structure

Installing DSE creates a number of directories on the host machine. This directory structure contains important files and information, such as message files, product documentation, working examples, and shared libraries.

The top-level directory of DSE is the directory in which the product is installed. This directory has the following subdirectories:

**bin** Contains the working binaries for DSE: rtl_make_pass, rtlmode, rtloader, rtlshmdmp, and rtlstat.

**doc** Contains the product documentation.

**examples** Contains working examples of a feed handler, a feed simulator, a client program, and DB2 user-defined functions. The examples subdirectory has the following subdirectories:

- **generic**
- **ha**
- **restart**
- **simple**

The examples subdirectory also contains the rtlconfig.template file, which contains all of the configuration parameters without preset values. You can use this file as a template to create your configuration file.

**generic subdirectory**

Has the following subdirectories:

- **config** Contains a sample configuration file.
- **driver** Contains the source code for the send_ticks program, a simple feed simulator.

**handler**

Contains the source code for a generic feed handler that handles different data types.

**setup** Contains the db2-schema.sql script file that you use create the ticks and symbols tables as part of setting up the DB2 database system, to prepare the DB2 database system to work with DSE.

**ha subdirectory**

Has the following subdirectories:

- **config** Contains a sample configuration file.
- **driver** Contains the source code for the send_ticks program, a simple feed simulator.

**handler**

Contains the source code for a high availability feed handler.

**include** Contains the header files that are needed by the source code in the examples/ha subdirectories.
setup  Contains the scripts that you use to set up the DB2
database system and the wrapper, to prepare the DB2
database system to work with DSE. These scripts include
the db2-schema.sql script to create the ticks and symbols
tables, and the wrapper_schema.sql script to create the
wrapper.

restart subdirectory
  Has the following subdirectories:
  config  Contains a sample configuration file.
  driver  Contains the source code for the mqsend program, a
          WebSphere MQ-based data feed simulator.
  include  Contains the header files that are needed by the source
           code in the examples/restart subdirectories.
  mqm  Contains sample WebSphere MQ configuration files.

simple subdirectory
  Has the following subdirectories:
  config  Contains sample configuration files.
    • The rtlconfig.teststoresys configuration file uses the
test storage system that is included with DSE.
    • The rtlconfig.db2 configuration file uses the DB2
database system as the storage system. The store in this
configuration uses a master-detail relationship.
    • The rtlconfig.db2_simple_schema configuration file uses
the DB2 database system as the storage system. The
store in this configuration uses a simple schema.
  driver  Contains the source code for the send_ticks program, a
simple feed simulator. The send_ticks program works for
stores using a master-detail relationship or a simple
schema.
  handler  Contains the source code for a simple feed handler. The
simple feed handler works for stores using a master-detail
relationship or a simple schema.
  include  Contains the header files that are needed by the source
code in the examples/simple subdirectories.
  setup  Contains the scripts that you use to set up the DB2
database system and the wrapper, to prepare the DB2
database system to work with DSE. These scripts are:

    db2-schema.sql
      To create the ticks and symbols tables. Use this
      script if you are using the rtlloader.db2
      configuration file for data in a master-detail
      relationship.

    db2-schema_simple_schema.sql
      To create the ticks table for a store using a simple
      schema. Use this script if you are using the
rtlconfig.db2_simple_schema configuration file for data using a simple schema.

**sample_wrapper.sql**
To create the wrapper. Use this script if you are using the rtloader.db2 configuration file for data in a master-detail relationship.

**simple_apps**
Contains an example client program that uses the DSE C-API.

**udf**
Contains examples of DB2 user-defined functions that interact with DSE.

**include**
Contains the public header files for DSE.

**java**
Contains the jar files and javadocs for the Java API. The java directory contains two subdirectories:
- **doc** for the javadocs. The javadocs contains information about the Java API.
- **lib** for the libraries.

**lib**
Contains the shared libraries that DSE uses.

**license**
Contains the license information for your locale.

**logs**
Contains the installation logs.

**msg**
Contains the message files used by DSE.

**storesys**
Contains an example storage system implementation. This example is a simple test storage system that has no real functionality. This test storage system simply returns appropriate values to indicate success for each of the storage system interface functions.

**tech_preview**
Contains unsupported technical previews. See the documentation that is provided in this subdirectory.

The top-level directory of DSE also contains the makeinc file, which contains platform-specific values that DSE uses when it compiles source code in the examples directory.
Chapter 6. Tutorial: Getting started with DSE

This tutorial demonstrates how to get up and running with DSE, by describing how to set up a storage system and run the simple example to verify the DSE installation.

Before using this tutorial, you must install DSE and DB2 Version 9.1 (with Fix Pack 3) or DB2 Version 9.5.

The first part of the tutorial involves setting up a test storage system. The second part involves verifying the installation of DSE using one of two methods:

**Verifying your installation using the quick start script**
This method uses a script to automatically run the steps of the simple example.

**Verifying your installation step by step**
This method demonstrates how to run each step of the simple example manually, to help you understand how to set up and operate a basic DSE system.

With both methods, you verify the installation of DSE by building a simple data feed and feed handler, running DSE to load that data into a test DB2 database, and querying the data in the database.

### Setting up a test storage system

Set up a storage system, such as the DB2 database system, as a prerequisite step to verifying the installation of DSE. The installation verification process involves loading sample data into a DB2 database.

If you are using a DB2 database as your test database, complete the following tasks:
- "Preparing the DB2 instance"
- "Setting up the environment variables” on page 18

If your database server is on a machine that is separate from DSE, you need to configure the client server communications, including cataloging the remote database on the DSE machine.

### Preparing the DB2 instance

As part of setting up a DB2 test database to verify the DSE installation, prepare the DB2 instance.

DB2 Version 9.1 (with Fix Pack 3) or DB2 Version 9.5 must be installed.

To prepare the DB2 instance:
1. Create a DB2 instance.
2. If you are using the DSE wrapper, register the product license key for WebSphere Federation Server by running the following command:

   `db2licm -a wsfs_t.lic`
3. Turn DB2 federation on. This task is required if you plan to use the DB2 wrapper to query data or if you use the rtl_quick_start script to verify the DSE installation.
   a. Stop the DB2 instance.
   b. Run the following command:
      
      ```
      db2 update dbm cfg using federated yes
      ```
   c. If you installed the DSE wrapper in [Chapter 4, “Installing DSE,” on page 11](http://publib.boulder.ibm.com/infocenter/db2luw/v9r5), complete its installation by completing the following steps:
      1) Become the root user.
      2) Set and export the INSTHOME environment variable to the home directory of the DB2 instance owner.
      3) Run the appropriate wrapper installation script. This file is located in the directory in which DSE is installed.

      ```
      wrapper_install_v913.RUN_AS_ROOT
      ```
      If the version of the DB2 instance for the federated server is DB2 Version 9.1 (Fix Pack 3).

      ```
      wrapper_install_v95.RUN_AS_ROOT
      ```
      If the version of the DB2 instance for the federated server is DB2 Version 9.5.

4. Start the DB2 instance.
5. Create a DB2 test database and name it TESTDB. You will use this database when you verify your installation.

For information about using the DB2 database system, including information about creating a DB2 instance, see the DB2 Version 9.5 information center at [http://publib.boulder.ibm.com/infocenter/db2luw/v9r5](http://publib.boulder.ibm.com/infocenter/db2luw/v9r5) or the DB2 Version 9.1 information center at [http://publib.boulder.ibm.com/infocenter/db2luw/v9](http://publib.boulder.ibm.com/infocenter/db2luw/v9)

### Setting up the environment variables

Set up the DB2 environment before you can verify the installation DSE.

To set up the DB2 environment:

1. Run the DB2 setup file to set up the environment variables that are required for the DB2 environment (INSTHOME, DB2DIR, and DB2INSTANCE).
   • For a Bourne or Korn shell, run the following command:
     ```
     . ~$db2inst/sqllib/db2profile
     ```
     Where `db2inst` is the name of the DB2 instance that you are using.
   • For a C shell, run the following command:
     ```
     source ~$db2inst/sqllib/db2cshrc
     ```
     Where `db2inst` is the name of the DB2 instance that you are using.

2. In any directory on your system, create a new file and add any additional environment variables that you need to it. The environment variable MACHINE is required. The remaining environment variables are useful to verify your installation, but are not required. The following example shows the contents of a sample environment variables file for someone using a Bourne or Korn shell. This example does not apply when using a C shell:

   ```
   export MACHINE=linux
   export DSE_INSTALL_DIR=/opt/ibm/dse/v9.5.0
   export LD_LIBRARY_PATH=$DSE_INSTALL_DIR/lib:$LD_LIBRARY_PATH
   export PATH=$DSE_INSTALL_DIR/bin:$PATH
   ```

   In this example:
• Replace the value for the MACHINE variable with a value specific to your platform:
  – MACHINE=aix64 (for AIX)
  – MACHINE=linux (for Linux 32-bit)
  – MACHINE=linux64 (for Linux 64-bit)
  – MACHINE=solaris64 (for Solaris)
• Replace the value for the DSE_INSTALL_DIR variable with the directory in which DSE is installed.

3. Source the environment variables file.

Verifying your installation with the quick start script

DSE provides a script to quickly configure and run DSE, to verify its installation.

Before you run this script, you must complete the following tasks:
• Chapter 4, “Installing DSE,” on page 11
• Installing DB2 Version 9.1 (with Fix Pack 3) or DB2 Version 9.5
• “Preparing the DB2 instance” on page 17
• “Setting up the environment variables” on page 18

This script runs the simple example to quickly verify the installation of DSE. You can run this script or you can manually reproduce the steps that the rtl_quick_start script performs by completing the tasks in “Verifying your installation step by step.”

To quickly verify the installation of DSE:
1. Run the following command to start the script:
   
   ```bash
   $DSE_INSTALL_DIR/bin/rtl_quick_start.sh
   ```
2. The script prompts you for the following information:
   • The location of your DB2 instance (if you did not set and export the INSTHOME environment variable)
   • The name of the test database that you created when you prepared your DB2 instance
   • A port number for DSE to communicate with client applications
   • A port number for DSE to receive data from the feed simulator
   • A unique shared memory key
   • A store schema (master-detail or simple)
3. The script configures DSE and runs the simple example to verify the DSE installation.
   • If DSE ran successfully, the message “DSE ran successfully” is displayed.
   • If the script encountered a problem, it displays an error message and logs the message in the quick_start.log file in the $DSE_INSTALL_DIR/logs directory. Check the error log to determine the cause of the problem.

Verifying your installation step by step

DSE includes a simple example to help you get started with using the product. The simple example includes sample programs, configuration files, and SQL files in subdirectories of the examples/simple subdirectory. You can use parts of the simple example to verify your installation.
To verify your installation:
1. Set up a store.
2. Build a simple data feed simulator and feed handler.
3. Configure DSE.
4. Start DSE.
5. Set up the DSE wrapper and run a query to confirm that the wrapper is set up correctly.

For a description of the contents of the examples/simple subdirectory, see Chapter 5, “DSE directory structure,” on page 13.

Setting up a store

Set up the store to which DSE flushes data. If you plan to use the test storage system that is included with DSE to verify your installation, you can skip this task.

You can set up a store using a master-detail relationship or a simple schema.

DB2 databases: Tables and feeds

DSE stores data that is in a master-detail relationship in two tables in a DB2 database; these two tables comprise a store. DSE stores data that uses a simple schema in a single table in a DB2 database; this table contains ticks data.

A store that uses the master-detail relationship contains the following two tables:

Symbols table
This table stores symbols data and symbols metadata. The STOREXX_SYMBOL_TABLE parameter defines the name of this table.

Ticks table
This table stores ticks data. The STOREXX_TICK_TABLE parameter defines the name of this table.

The symbols table and ticks table share the same schema name. The STOREXX_SCHEMA parameter specifies this schema name.

To use a DB2 database as the storage system for master-detail data, create these tables for each feed within your database. To use a DB2 database as the storage system for simple-schema data, create a single table for each feed within your database.

You can have each feed insert data into a separate table, or you can have multiple feeds load data into the same table. You can specify a different database table for DSE to load for a specific feed by defining the FEEDXX_STORE_NAME parameter (where XX is the number of the feed) in the configuration file.

The send_ticks program simulates a feed. DSE can be configured to store the feed in a master-detail relationship or a simple schema.

Creating DB2 tables

DSE provides sample SQL files that you can use to create generic DB2 tables. Use the db2-schema.sql file to create two tables for data in a master-detail relationship. Use the db2-schema_simple_schema.sql file to create a single table for data in a simple schema.

To create these tables:
1. Ensure that a test database named TESTDB is created.
2. Locate the db2-schema.sql or db2_schema_simple_schema.sql file in the examples/simple/setup subdirectory. The db2-schema.sql file creates a symbols table (rtl.ticks_symbols), a ticks table (rtl.ticks_ticks), and an index for the ticks table. You can use the db2-schema.sql file as provided for the purposes of verifying your installation, or you can edit it and use it to help you create your own tables.
3. Optional: Edit the SQL file to create your own tables. The structure of the table is as follows. The columns in the tables can be in any order. The names of the columns must match the column names in the RECORDXX_SCHEMA parameter in your configuration file.

   **The symbols table**
   The symbols table must have at least two columns. One of the columns must be called symbol_name and be of type VARCHAR(32) NOT NULL. The other column must be called symbol_id and be of type INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY. The symbols table can have additional columns.

   **Requirement:** For high availability, the symbol_id column must have a unique index.

   **The ticks table**
   The ticks table must have a timestamp column. This table must also have a column that is called symbol_id and is of type INTEGER NOT NULL. The ticks table can have additional columns that are not found in the RECORDXX_SCHEMA parameter.
4. Run the db2-schema.sql or db2_schema_simple_schema.sql file against the TESTDB database.
5. Optional: If you want to run the db2-schema.sql or db2_schema_simple_schema.sql file against a different database, modify the file and replace TESTDB with the name of the database that you are using.

**Building a simple data feed simulator and feed handler**

To verify the installation, build a simple data feed simulator and feed handler. The send_ticks program is included with the product and simulates a simple data feed. The simple feed handler is included with the product.

To complete this task:
1. Build the simple data feed simulator by running the following commands:
   ```
   cd $DSE_INSTALL_DIR/examples/simple/driver
   make
   ```
   **Note:** Use gmake instead of make on the AIX and Solaris platforms.
2. Build the simple feed handler by running the following commands:
   ```
   cd $DSE_INSTALL_DIR
   make
   ```
   When you build the simple feed handler, the simplehandler.so library file is created and saved in the $DSE_INSTALL_DIR/lib subdirectory.
3. Start the send_ticks program in the background by entering the following command:
   ```
   $DSE_INSTALL_DIR/examples/simple/driver/send_ticks -p3001 \ 
   -f$DSE_INSTALL_DIR/examples/simple/driver/tick_data \
   ```
In this example, 3001 is the port number that represents the port from which the feed is set to receive data. The port number must match the end of the FEEDXX_SOURCE parameter value in the configuration file that you use with DSE. For example, the value of the FEED01_SOURCE parameter in the rtlconfig.db2 file (in the examples/simple/config subdirectory) is "127.0.0.1:3001," so the port number is 3001.

**Configuring DSE to verify the installation**

Create a configuration file before starting DSE to verify the installation.

**Prerequisites:** Build a simple data feed and feed handler. See "Building a simple data feed simulator and feed handler" on page 21 for information.

DSE provides the following sample configuration files:

**rtlconfig.teststoresys**
This configuration file uses the test storage system that is included with DSE. You can use this file to verify your installation without having a real storage system in place.

**rtlconfig.db2**
This configuration file uses a DB2 database for the storage of data in a master-detail relationship. If you use this configuration file, you must have a DB2 test database with the correct schema. See "DB2 databases: Tables and feeds" on page 20 for instructions for using an included SQL file to create generic DB2 tables.

**rtlconfig.db2_simple_schema**
This configuration file uses a table in a DB2 database for the storage of data in a simple schema. See "DB2 databases: Tables and feeds" on page 20 for instructions for using an included SQL file to create a generic DB2 table.

To create your own configuration file based on a sample configuration file:

1. Create a copy of the appropriate sample configuration file and give the new file a name. For example, to copy the rtlconfig.db2 file and name the new file rtlconfig.dse, enter the following command:

   ```bash
cp $DSE_INSTALL_DIR/examples/simple/config/rtlconfig.db2 \   $DSE_INSTALL_DIR/examples/simple/config/rtlconfig.dse
   
   If you want to customize your configuration file, see Chapter 9, "System configuration parameters," on page 51 to see detailed information about the configuration parameters. An IBM consultant can help you set up DSE.

2. Edit your copy of the configuration file and replace the following parameter values with ones that are appropriate for your system:

   **SHMKEY**
   Replace DSE1_SHMKEY with an integer that represents a unique shared memory key to be used (for example, 268658432).

   **RTL_SERVICE_NAME**
   Replace DSE1_PORT with the port number to which a client application (such as the wrapper) connects (for example, 9324).

   **Restrictions:**
   - This number must be different from the value of DSE_TICKS_BASE_PORT.
Starting DSE to verify the installation

To verify the installation, ensure that you can start DSE without any errors.

To start DSE:

1. Start DSE by entering the following command:
   
   ```
   $DSE_INSTALL_DIR/bin/rtloader -c \
   $DSE_INSTALL_DIR/examples/simple/config/rtlconfig.dse &
   ```

   **Note:** This command starts DSE in the background with the rtlconfig.dse configuration file that you just created.

2. Confirm that you started DSE without any errors by checking the contents of the log file. If the log file has no errors, you have successfully started DSE. The RTL_LOG_PATH parameter in the configuration file defines the log file that is used. For example, the rtlconfig.db2 file defines the log file as rtl.log:
   
   ```
   RTL_LOG_PATH  "rtl.log"
   ```

   In this case, you can find rtl.log in the examples/simple/config directory. In another example, you can define the log file using an absolute path in your configuration file:
   
   ```
   RTL_LOG_PATH  "/devel/rtl.log"
   ```

   In this case, you can find the rtl.log file in the /devel directory. Specify an absolute path for the log file in the configuration file if you prefer to put the log file in a specific directory.

   For more information about the rtloader command and its options, see Chapter 7, "rtloader command: Starting DSE," on page 29.

Setting up a wrapper to query a DB2 database

DSE includes a wrapper that you can install and set up. A wrapper enables you to issue SQL statements in a DB2 database to get data from external sources, such as a remote DSE server.

If you installed the wrapper, set up the wrapper and run a simple query to ensure that you can retrieve data that DSE flushed to the DB2 database. These steps use the simple example that is provided by DSE to demonstrate how to set up a wrapper and run a data query. You can create and customize your own wrapper, or an IBM consultant can help you.

**Prerequisites:** Ensure that you completed the following steps:

1. Install the DSE wrapper using the install_dse program. See Chapter 4, "Installing DSE," on page 11 for instructions.
2. Create a DB2 instance and turn federation on. See "Setting up a test storage system" on page 17 for instructions.

3. Set up a storage system. See "Setting up a store" on page 20 for instructions.

4. Start the sample send_ticks program. See "Building a simple data feed simulator and feed handler" on page 21 for instructions.

5. Start DSE. See "Starting DSE to verify the installation" on page 23 for instructions.

To set up your DSE wrapper, you must create and work with the following DB2 database objects:

**Wrapper**

Provides the mechanism for the DB2 database to interact with a remote data source, such as a DSE server.

**Server**

Defines a data source for a wrapper.

**User mapping**

Maps a DB2 user to a remote server.

**Nickname**

Maps a local DB2 object to a remote table or view.

For more information about wrappers, see the following information:

- **Federated Systems Guide**

  This information is available for download from www.ibm.com/software/data/db2/library.

- **DB2 UDB SQL Reference Volume 2**

  This information documents the CREATE WRAPPER, CREATE SERVER, CREATE USER MAPPING, and CREATE NICKNAME commands. This information is available for download from www.ibm.com/software/data/db2/library.

A sample file, sample_wrapper.sql, is included in the examples/simple/setup subdirectory and includes examples of SQL statements to perform steps that are similar to the steps in this topic.

To set up the wrapper that is provided by DSE:

1. In the sample_wrapper.sql file, replace the value of the DSE1_PORT parameter. Provide a number that represents the port to which the client (such as a wrapper) connects (for example, 9324).

   **Restrictions:**

   - This number cannot match the value of the DSE_TICKS_BASE_PORT parameter.
   - This number cannot be used by any other application.

2. Connect to your DB2 test database, TESTDB.

3. Create the DSE wrapper in the TESTDB database. Enter the following DB2 command:

   ```
   create wrapper DSE library 'librtl.so' options (DB2_FENCED 'N')
   ```

   If you want to turn trace on for debugging, you can enter the following command instead. This command specifies an external trace file for messages and a trace level, to give you the maximum trace detail for debugging problems.

   ```
   create wrapper DSE library 'librtl.so' options (DB2_FENCED 'N',
   TRACE_FILE '/tmp/rtl.trace', TRACE_LEVEL '40')
   ```
You can give the wrapper any name that you want. In these sample commands, the wrapper name is DSE.

4. Create the database server that queries data from the remote DSE server. Enter the following DB2 command:

   ```
   create server TSDATA wrapper DSE options(host_name 'localhost',
   service_name '9324')
   ```

   In this command:
   - The value 9324 matches the value of the DSE1_PORT parameter.
   - host_name specifies the host name for the remote DSE server. If the database server and remote DSE server are on the same host, you can use 'localhost' for host_name.

5. Create a user mapping, which stores authentication information for the remote DSE server. The wrapper uses this information to connect to that server. Enter the following DB2 command:

   ```
   create user mapping for your_user server TSDATA options (remote_authid 'testdb',
   remote_password 'marshmello')
   ```

   In this command:
   - The sample group ID testdb is defined by the GROUP01_NAME parameter in the examples/simple/client/rtlconfig.dse configuration file that you created in “Configuring DSE to verify the installation” on page 22.
   - The password marshmello corresponds to the encrypted string in the GROUP01_PASSWORD parameter in the examples/simple/client/rtlconfig.dse configuration file that you created in “Configuring DSE to verify the installation” on page 22.
   - The user mapping is associated with the database server TSDATA that you created in step 4. To debug the user mapping, create the user mapping for TSDATA_Debug instead of TSDATA.

   You can create your own GROUPXX_NAME and GROUPXX_PASSWORD using the rtl_make_pass utility.

   The DB2 database encodes the value for the remote_password option and stores it in the system catalogs in an unreadable form.

6. Create a DB2 nickname called tick_data and map it to the sample tick data structure. Your remote DSE server must be running before you create the nickname.

   a. Create a new SQL file and name it nicknametick.sql. Add the following SQL statement to the file:

   ```
   create nickname tick_data
   (symbol varchar(32),
   tstamp timestamp,
   trdprc_1 double,
   bid double,
   ask double,
   irgprc double,
   insprc double,
   trdvol_1 int,
   acvol_1 int,
   num_moves int,
   bidsize int,
   asksize int,
   irgvol int,
   insvol int,
   cleansethisfield int,
   act_flag char(2),
   irgcond char(4),
   inscond char(4))
   ```
for SERVER TSDATA
options(storagesys_name 'db2', store_name 'rtl.ticks');

You can find similar statements in the included sample_wrapper.sql file in the examples/simple/setup subdirectory.

Requirements:
• The first two columns in the nickname are required, because the DSE wrapper uses them to optimize predicates in the SQL WHERE clause. You can give these columns any name. In this example, the column names are symbol and tstamp. The SQL data types for these columns must be defined as follows:
  – symbol varchar(32)
  – tstamp timestamp
• The column names in the nickname must match the column names in the RECORDXX_SCHEMA definition exactly. If your column names do not match exactly, use the SRC_COL_NAME option to map your nickname column names to the column names in the RECORDXX_SCHEMA definition. See [“Options” on page 145] for more information about the SRC_COL_NAME option.
• The values for storagesys_name and store_name in this file must match the values that are defined for the STORAGESYS01_NAME and STORE01_NAME parameters in the examples/simple/client/rtlconfig.dse configuration file that you created in [“Configuring DSE to verify the installation” on page 22].

b. Save and close the nickname.tick.sql file.
c. Run the nickname.tick.sql file against the TESTDB database.

7. Run a simple query to ensure that you can retrieve the data that DSE flushed to the TESTDB database.

a. Create a new SQL file and name it query.tick.sql. Add the following SQL statement to the file:
   ```sql
   select * from rtl.ticks_ticks
   where tstamp between '2001-02-05-14.32.19.0005' and
   '2001-02-05-14.32.20.0005'
   and symbol='2194'
   order by 2;
   ```
   
   **Important:** Be sure that the column names in this file (tstamp and symbol) match the column names in your nickname.tick.sql file in step 6 on page 25.

b. Run the query.tick.sql file against the TESTDB database. If the query is successful, the requested data displays. This query retrieves all tick data that has a time stamp between the specified time stamps for the specified symbol.

By running this query, you demonstrate that the wrapper is set up correctly and that you are able to query data from your DB2 test database, TESTDB. The sample_wrapper.sql file that is included in the examples/simple/setup subdirectory includes additional examples of SQL statements to query data.
Part 3. Administering DSE

Administration tasks include configuring, operating, monitoring, and tuning DSE. DSE provides tools and samples to help you perform these tasks.
Chapter 7. rtloader command: Starting DSE

Use the **rtloader** command to start DSE and to control how it operates.

The **rtloader** command syntax is as follows:

```
```

The command options are described in the following table.

**Table 2. rtloader command options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c <code>config_file</code></td>
<td>Specifies the configuration file that DSE uses.</td>
</tr>
<tr>
<td></td>
<td>If you do not specify the <code>-c</code> option, DSE looks for a configuration file</td>
</tr>
<tr>
<td></td>
<td>named <code>rtlconfig</code> in the directory where you ran the <code>rtloader</code> command.</td>
</tr>
<tr>
<td>-e <code>error_file</code></td>
<td>Specifies the error message file to use.</td>
</tr>
<tr>
<td></td>
<td>This option overrides the <code>RTL_SYS_MSG_PATH</code> configuration parameter. For</td>
</tr>
<tr>
<td></td>
<td>more information about this parameter, see “General system configuration”</td>
</tr>
<tr>
<td></td>
<td>parameters” on page 52.</td>
</tr>
<tr>
<td>-f</td>
<td>Forcibly reinitializes shared memory.</td>
</tr>
<tr>
<td></td>
<td>To prevent multiple DSE instances from using the same area in shared</td>
</tr>
<tr>
<td></td>
<td>memory, DSE maintains an in-use flag. When DSE shuts down normally, this</td>
</tr>
<tr>
<td></td>
<td>flag is cleared. If DSE abnormally terminates, the flag might remain</td>
</tr>
<tr>
<td></td>
<td>uncleared.</td>
</tr>
<tr>
<td></td>
<td>Use the <code>-f</code> option to ignore the flag and reuse shared memory. Use this</td>
</tr>
<tr>
<td></td>
<td>option with caution; another DSE instance might be running and using shared</td>
</tr>
<tr>
<td></td>
<td>memory. You can check which shared memory keys are already assigned by</td>
</tr>
<tr>
<td></td>
<td>running the <code>ipcs -m</code> command.</td>
</tr>
<tr>
<td>-M l</td>
<td>s</td>
</tr>
<tr>
<td>-M l</td>
<td>DSE is started in live mode, and goes into ACTIVE state.</td>
</tr>
<tr>
<td>-M s</td>
<td>DSE is started in standby mode, and goes into STANDBY state after</td>
</tr>
<tr>
<td></td>
<td>synchronizing with its peer DSE that is in ACTIVE state.</td>
</tr>
<tr>
<td></td>
<td>You can start DSE in STANDBY state only if the HA_ENABLE configuration</td>
</tr>
<tr>
<td></td>
<td>parameter is set to 1.</td>
</tr>
<tr>
<td></td>
<td>The current state is displayed in the <code>rtlstat</code> command and can also be</td>
</tr>
<tr>
<td></td>
<td>determined with the <code>rtl_state_get</code> function.</td>
</tr>
</tbody>
</table>

**Related reading:** For more information, see “High availability states” on page 95.
### Table 2. `rtloader` command options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-n</code></td>
<td>If you specify <code>-n</code>, error messages are output to stderr instead of to the log file that is specified by the RTL_LOG_PATH parameter in the rtlconfig file.</td>
</tr>
<tr>
<td><code>-s shmkey</code></td>
<td>Specifies the shared memory key that DSE uses. The order of precedence for specifying the shared memory key is as follows, from highest to lowest: 1. Command line (using the <code>-s</code> option) 2. RTLSHMKEY environment variable 3. SHMKEY configuration parameter in the configuration file 4. Default value (268 658 176)</td>
</tr>
<tr>
<td><code>-t trace_dir</code></td>
<td>Specifies the directory that contains the trace file. This option overrides the TRACEFILE_DIR configuration parameter. For more information about this parameter, see “Trace configuration parameters” on page 79</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Displays version information for DSE.</td>
</tr>
<tr>
<td><code>-x</code></td>
<td>Reads the configuration file and exits without initializing DSE.</td>
</tr>
<tr>
<td><code>--help</code></td>
<td>Displays help information for this command.</td>
</tr>
</tbody>
</table>

**Note:** DSE ignores the following signals:  
- SIGABRT (Ctrl+\)  
- SIGALRM  
- SIGHUP  
- SIGQUIT  
- SIGINT (Ctrl+C)  
- SIGTSTP (Ctrl+Z)  
- SIGUSR2
Chapter 8. Utility programs

DSE provides command-line utility programs to aid in daily operations and troubleshooting.

DSE includes these command-line utility programs:
- **rtldiscarddmp**
  Converts data in a discard table into a delimited ASCII (DEL) file.
- **rtlmode**
  Changes the operating mode of DSE and performs various operations on data feeds and data stores.
- **rtlshmmdmp**
  Prints the information in the shared memory used by DSE, to help troubleshoot problems.
- **rtlstat**
  Monitors the operation and performance of DSE.
- **rtl_make_pass**
  Creates encrypted passwords for use in the configuration file.

### rtldiscarddmp utility

The rtldiscarddmp utility converts data in a discard table into a delimited ASCII (DEL) file. Use this utility as part of the process of correcting and reloading bad data.

The output DEL file can be used with the DB2 import utility to reload data into DB2 after correcting the data errors.

The rtldiscarddmp command syntax is:
```
rtldiscarddmp -c config_file [ -d delimiter ] -s store_id [ -r row_range ]
-o output_file
```

The command options are described in the following table.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c config_file</td>
<td>Specifies the absolute path of the DSE configuration file.</td>
</tr>
<tr>
<td>-d delimiter</td>
<td>Optional. Specifies the column delimiter. If the delimiter is not specified, a comma (,) is used.</td>
</tr>
<tr>
<td>-s store_id</td>
<td>Specifies the store ID.</td>
</tr>
<tr>
<td>-r row_range</td>
<td>Specifies the range of rows, based on user-provided tick ID values, from the discard table to be converted into delimited format in the DEL file. For example, -r50,100 indicates that the rows with tick IDs 50 through 100 in the discard table will be converted.</td>
</tr>
</tbody>
</table>
Table 3. rtdiscarddmp command options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o output_file</td>
<td>Specifies the name of the output DEL file.</td>
</tr>
</tbody>
</table>

The output DEL file has the following characteristics:
- NULL values are represented by an empty space between the column delimiters.
- Character strings are enclosed in quotation marks (" ").
- Values containing quotation marks are escaped by a quotation mark. For example, the value a"b"c is represented as "a""b""c" in the output DEL file.
- The value "NAN" (when used for a numeric column) means that the value cannot be interpreted properly according to the data type that is specified in the record schema.
- Each line in the DEL file corresponds to a row in the discard table.

The rtdiscarddmp utility is written as a DB2 CLI application.

Related concepts
Chapter 20, “Correcting and reloading bad data,” on page 133
Data feeds can contain data that is badly formed or that does not comply with data constraints. When DSE detects this bad data while flushing to the target store, DSE discards the bad data into a discard table. You can then analyze and correct the data, and then reload it into the target ticks table.

rtlmode utility

The rtlmode utility controls DSE. It changes the operating mode of DSE and performs various operations on data feeds and data stores.

The rtlmode command syntax is:
```bash
rtlmode [option] [ -s shared_memory_key ] [ -y ]
```

where option is at most one of the options that are described in the following table.

Table 4. rtlmode command options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a {e</td>
<td>d}{store_id</td>
</tr>
<tr>
<td></td>
<td>• Use e to enable automatic writes of metadata.</td>
</tr>
<tr>
<td></td>
<td>• Use d to disable automatic writes of metadata.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “-a option: Enable automatic writes of metadata to the store” on page 34.</td>
</tr>
</tbody>
</table>
### Table 4. \texttt{rtlmode} command options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| \texttt{-d \{e|d\}\{store\_id\}all} | Enables or disables disk writes to the specified store.  
  \begin{itemize}  
  \item Use e to enable disk writes.  
  \item Use d to disable disk writes.  
  \end{itemize}  
  For more information, see \texttt{-d option: Enable or disable disk writes" on page 35.} |
| \texttt{-D [START|STOP]} | Starts or stops writing trace information to the trace file. If neither START nor STOP is specified, DSE dumps the current trace buffer contents to the trace file.  
  For more information, see \texttt{-D option: Start or stop writing trace information to the trace file" on page 35.} |
| \texttt{-f \{d|r|p|s|t\}\{feed\_number\} | Changes the feed source for a specific feed or all of the feeds.  
  \begin{itemize}  
  \item \texttt{l} new\_source \texttt{| new\_target \texttt{| delete\_source}}  
  \end{itemize}  
  For more information, see \texttt{-f option: Control feeds" on page 35.} |
| \texttt{-F \{e|d\}\{feed\_number\}all} | Enables or disables the DSE function to always accept ticks and add them to shared memory, regardless of whether the data store is online or offline, for the specified feed or for all feeds.  
  For more information, see \texttt{-F option: Always add ticks to shared memory" on page 36.} |
| \texttt{-k} | Shuts down DSE after waiting for flushing of shared memory to complete.  
  For more information, see \texttt{-k option: Flush data and shut down DSE" on page 37.} |
| \texttt{-M \{l|s\}} | Sets the state to ACTIVE or STANDBY.  
  \begin{itemize}  
  \item l Set the state to ACTIVE.  
  \item s Set the state to STANDBY.  
  \end{itemize}  
  For more information, see \texttt{-M option: Change the high availability state" on page 37.} |
| \texttt{-o \{store\_id\}all} | Takes the specified store offline. Use the word all to take all stores offline.  
  For more information, see \texttt{-o option: Taking the store offline" on page 37.} |
| \texttt{-r} | Causes DSE to reread the configuration file. Use this option when you change select parameters.  
  For more information, see \texttt{-r option: Change select parameters dynamically" on page 37.} |
Table 4. \texttt{rtlmode} command options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{-T component[.{trace_level}]}</td>
<td>Displays or changes the trace level setting for the specified component.</td>
</tr>
<tr>
<td>\hspace{1em}For more information, see &quot;\texttt{-T option: Display or change the trace level setting for the specified component}&quot; on page 36.</td>
<td></td>
</tr>
<tr>
<td>\texttt{-u store_id</td>
<td>all}</td>
</tr>
<tr>
<td>\hspace{1em}For more information, see &quot;\texttt{-u option: Bring the store online}&quot; on page 39.</td>
<td></td>
</tr>
<tr>
<td>\texttt{-v}</td>
<td>Displays version information for DSE.</td>
</tr>
<tr>
<td>\texttt{-w store_id</td>
<td>all}</td>
</tr>
<tr>
<td>\hspace{1em}For more information, see &quot;\texttt{-w option: Initiate a manual write of metadata to the store}&quot; on page 39.</td>
<td></td>
</tr>
<tr>
<td>\texttt{-x}</td>
<td>Shuts down DSE immediately without flushing shared memory.</td>
</tr>
<tr>
<td>\hspace{1em}For more information, see &quot;\texttt{-x option: Shut down DSE immediately}&quot; on page 40.</td>
<td></td>
</tr>
<tr>
<td>\texttt{--help}</td>
<td>Displays help information for this command.</td>
</tr>
<tr>
<td>\texttt{-s shared_memory_key}</td>
<td>Specifies the shared memory key that was used when you started DSE with the \texttt{rtloader} command.</td>
</tr>
<tr>
<td>\hspace{1em}Note: The -s option can be combined with other \texttt{rtlmode} options.</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em}For more information, see &quot;\texttt{-s option: Specify a non-default shared memory key}&quot; on page 40.</td>
<td></td>
</tr>
<tr>
<td>\texttt{-y}</td>
<td>Turns off the requirement for confirmation for the -a, -k, -x, and -w options.</td>
</tr>
</tbody>
</table>

\textbf{-a option: Enable automatic writes of metadata to the store}

Use the -a option to enable or disable automatic writes of metadata from shared memory to a specific store or to all stores that use a master-detail relationship. This option is not supported for stores that use a simple schema.

- e DSE automatically writes metadata from shared memory to the specified store or to all stores.
- d DSE does not automatically write metadata from shared memory to any store.

You can specify a specific store ID or enter all. The store ID corresponds to the XX value in the store configuration parameters, where XX is a numeric value between 0 and 99.
**Requirement:** Do not enter a space between the e or d sub-options and the store_id number or the all word.

This command overrides the STOREXX_META_MODE parameter, if STOREXX_META_MODE=1 or STOREXX_META_MODE=2.

If STOREXX_META_MODE=0, issuing this command results in an error, because you might overwrite valid data that is in the store with invalid data that is in shared memory. The following error message displays:

Error: Metadata reads and writes for store_id are disabled.

**-d option: Enable or disable disk writes**

Use the -d option to enable or disable disk writes to a specific store or to all stores. Issuing this command is the equivalent of changing the value of the STOREXX_NODISKWRITE configuration parameter while DSE is running.

**-D option: Start or stop writing trace information to the trace file**

This command starts or stops writing trace information to the trace file.

```bash
rtlmode -D
```

DSE dumps the current trace buffer contents to the trace file.

```bash
rtlmode -D START
```

DSE starts writing trace information to the trace file. It stops when the `rtlmode -D STOP` command is run.

```bash
rtlmode -D STOP
```

DSE stops writing trace information to the trace file.

**-f option: Control feeds**

Use the -f option to modify settings related to feed sources. This option uses the following keywords:

- `feed_number`
  - Specifies the feed number in the configuration file. Use the keyword all with the p or r options to pause or resume all feeds.

- `new_source`
  - Specifies a new feed source. Repeated invocations of `rtlmode -fs new_source` create a waiting list of sources.

- `delete_source`
  - Specifies a pending source to be deleted from the waiting list.

- `new_target`
  - Specifies a new target to be switched to.

Use the following options to control the data feeds:

```bash
rtlmode -f delete_source
```

Delete source `delete_source` from the list of pending sources for feed `feed_number`.

```bash
rtlmode -f feed_number
```

List all pending sources for feed `feed_number`.

```bash
rtlmode -f p[feed_number | all]
```

Pause one or all feeds. In a paused state no messages are read from the
data source or written to a data target. Specify a feed number to be paused, or use the all keyword to pause all feeds.

```
rtlmode -f r[feed_number | all]
```
Resume one or all feeds. When resumed, a feed picks up where it was when it was suspended and reads the next message from the data source. Specify a feed number to be resumed, or use the all keyword to resume all feeds.

```
rtlmode -f sfeed_number new_source
```
Switch to a new source for feed feed_number. Depending on the type of the transport, the actual switch might happen immediately, or it might be deferred until the current source has been completely processed. In the latter case, the new source is added to a waiting list of sources that can be managed with the l and d options.

```
rtlmode -f tfeed_number new_target
```
Switch to new_target for feed feed_number. The actual switch occurs as soon as the feed finishes processing a message from the source.

Examples:

```
rtlmode -f pall
Suspends all feeds.
```
```
rtlmode -f r17
Resumes feed 17.
```
```
rtlmode -f s17 nyse_feed
Switches feed 17 to the nyse_feed source.
```
```
rtlmode -f l29
Displays a list of all pending feed sources for feed 29.
```
```
rtlmode -f d4 nasdaq_source
Deletes the source nasdaq_source from the list of pending sources from feed 4.
```
```
rtlmode -f t13 foo
Switches feed 13 feed to the new target foo.
```

The space between the -f switch and its argument is optional. No space is allowed between the p, r, s, l, d or t switches and the feed number.

**-F option: Always add ticks to shared memory**

Enables or disables the DSE function to always accept ticks and add them to shared memory.

```
e
```
DSE always accepts ticks and adds them to shared memory, regardless of whether the data store is online or offline.

```
d
```
DSE accepts ticks and adds them to shared memory only when the data store is online.

You can set this option for a specific data feed, or you can set it for all data feeds. Enter a specific feed number, or enter the word all, if you want to apply this option to all data feeds. The word all is not case-sensitive.

Examples:
rtlmode -F e5
   Adds ticks from feed 5 to shared memory, regardless of whether the data store is online or offline

rtlmode -F d6
   Adds ticks from feed 6 to shared memory only if the data store is online

rtlmode -F dall
   Adds ticks from all feeds to shared memory only if the stores are online

rtlmode -F eall
   Adds ticks from all feeds to shared memory, regardless of whether the data store is online or offline

-k option: Flush data and shut down DSE

When you use the -k option to shut down DSE, it first shuts down each feed handler, then waits for the new symbols in the store to be created. Then, DSE performs a full flush. After this flush completes, DSE shuts down. This process ensures that all of the data that is in shared memory is written to the store before DSE shuts down.

-M option: Change the high availability state

Use the -M option to change the high availability state of a DSE instance. See “High availability states” on page 95 for more information about high availability states.

Note: The rtlmode -Ms command is not currently supported. To change a DSE instance from active to standby, stop and restart DSE.

-o option: Taking the store offline

Use the -o option to take a specific store or all stores offline. Ticks that normally go into the store are discarded and are not placed in shared memory, if either of the following conditions is true:
- The FEEDXX_ACCEPT_TICKS_ALWAYS parameter is set to 0.
- The rtlmode -F d command was run and therefore, adding ticks to shared memory is disabled.

New symbols are not assigned to that store. Any ticks in memory for that store are not flushed, and remain in shared memory. All threads that are using the store disconnect from the storage system to which the store belongs.

-r option: Change select parameters dynamically

You can adjust the parameters listed below dynamically, by first changing them in the configuration file and then running the rtlmode -r command. DSE reads the configuration file again and makes the necessary changes. Changes to parameters other than these in the rtlconfig file are ignored until you restart DSE.

PURGE_HWM
   Specifies when DSE purges the tick pool HWM.

PURGE_LWM
   Specifies when DSE purges the tick pool LWM.

NODISKWRITE
   Specifies whether DSE writes the ticks to disk.
ENABLE_NEWSYM
   Specifies whether DSE can accept new symbols from the data stream.

-T option: Display or change the trace level setting for the specified component

Run the rtlmode -T {component}[.{trace_level}] command to display or change the trace level setting for the specified component.

rtlmode -T {component}
   This command displays the current trace level that is defined for the specified component.
   For example, the command rtlmode -T ALL displays the trace level setting for all of the components.

rtlmode -T {component}.{trace_level}
   This command changes the trace level setting of the specified component to the specified trace level.
   For example, issuing the command rtlmode -T ALL.med changes the trace level of all of the components to the medium trace level.

The component variable can have one of the following values. These values are case-sensitive.

ALL     All of the DSE threads and components.
FEED    The feed thread that processes the feed and invokes the feed handler library.
FLUSHER The flusher thread that writes ticks from memory to the persistent store.
HA      The high availability component of DSE.
HANDLER The user-defined feed handler library.
HA_COMM_LIB A user-defined high availability communication library.
INSERTER The inserter thread that manages new symbols in the store.
LISTENER The thread that processes rtlmode commands
MAIN    The main program, including startup and shutdown.
METAWRITER The metawriter thread that writes symbol metadata from memory to the persistent store.
PARTITIONER The partitioner thread that allocates ticks to flusher threads.
PURGER   The purger thread that removes old ticks from memory, to create space for new ticks.
QUERY   The query thread that processes API requests.
**STORESYS**

The storage system library.

**TRANSPORT**

The transport library that reads messages from different types of data sources.

The `trace_level` variable can have one of the following values. These values are not case-sensitive.

**High**  
High-level trace provides a high-level view of the execution of specified components. Of the trace levels, high-level trace produces the least amount of trace information. High-level trace information includes entry and exit points of the major functions within the specified components of DSE.

**Med**  
Medium-level trace provides a more detailed view of the functions that are run within the specified components. Medium-level trace produces more information than high-level trace and includes the information from high-level trace. Medium-level trace information includes significant information and states within major functions, and entry and exit points of lower-level functions within the specified components of DSE.

**Low**  
Low-level trace provides the most detailed review of the functions that are run within the specified components. Of the trace levels, low-level trace produces the most amount of information. Low-level trace information includes all trace information within the specified components of DSE. The volume of trace information that is written at this level might impact performance.

**Off**  
Trace is off.

**Examples:**

The following command disables trace for all of the components of DSE:

```
rtlmode -T ALL.OFF
```

The following command changes the trace level of the feed handler to the low setting:

```
rtlmode -T handler.low
```

**-u option: Bring the store online**

Use the `-u` option to bring a specific store or all stores online. All threads that are using the store reconnect to the storage system to which the store belongs.

**-w option: Initiate a manual write of metadata to the store**

This command initiates a manual write of metadata from shared memory to a specific store or to all stores that use a master-detail relationship. This option is not supported for stores that use a simple schema.

The `RECORDXX_META_SCHEMA` parameter must define the metadata field or fields with the `STORE READWRITE` or `STORE READWRITE AUTO` keywords, or the metadata is not written when this command is run. This command writes only the metadata in shared memory that has changed since the last manual write of metadata to the store.

You can specify a specific store name or enter all.
If STOREX_META_MODE=0, issuing this command results in an error, because you might overwrite valid data that is in the store with invalid data that is in shared memory. The following error message displays:

Error: Metadata reads and writes for store_id are disabled.

-x option: Shut down DSE immediately

When you use the -x option to shut down DSE, the usual mechanism of waiting for new symbols to be created and flushing to finish is bypassed and DSE is terminated immediately.

-s option: Specify a non-default shared memory key

Use the -s option to specify the shared memory key that was used when you started DSE.

The order of precedence for specifying the shared memory key is as follows, from highest to lowest:

1. Command line (using the -s option)
2. RTLSHMKEY environment variable
3. Default value (268 658 176)

rtlshmdmp utility

The rtlshmdmp utility prints the information in DSE shared memory, to help troubleshoot problems.

The rtlshmdmp command syntax is:

rtlshmdmp [option] [ -s shared_memory_key]

where option is one or more of the options that are described in the following table.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>Prints all of the information in shared memory.</td>
</tr>
<tr>
<td>-b</td>
<td>Prints the trace buffer contents.</td>
</tr>
<tr>
<td>-c</td>
<td>Displays the contents of the configuration file.</td>
</tr>
<tr>
<td>-C</td>
<td>Performs a consistency check of the information in shared memory, and reports any information that might be inconsistent. For stores that use a simple schema, the consistency check for symbols is not performed.</td>
</tr>
</tbody>
</table>
### Table 5. rtlshmdmp command options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d [fFhipstu]</code></td>
<td>Prints derived information. You can specify any number of the following options:</td>
</tr>
<tr>
<td><code>f</code></td>
<td>Prints distribution information about the flushers.</td>
</tr>
<tr>
<td><code>F</code></td>
<td>Prints distribution information about feed partitioning queues.</td>
</tr>
<tr>
<td><code>h</code></td>
<td>Prints summary distribution information about the hash buckets.</td>
</tr>
<tr>
<td><code>i</code></td>
<td>Prints distribution information about ticks across interval buckets. This output does not include information about stores that use a simple schema.</td>
</tr>
<tr>
<td><code>p</code></td>
<td>Prints distribution information about ticks that are flushed but are not yet purged.</td>
</tr>
<tr>
<td><code>s</code></td>
<td>Prints distribution information about ticks across symbols. This output does not include information about stores that use a simple schema.</td>
</tr>
<tr>
<td><code>t</code></td>
<td>Prints distribution information about the ticks.</td>
</tr>
<tr>
<td><code>u</code></td>
<td>Prints distribution information about unflushed ticks.</td>
</tr>
</tbody>
</table>

If none of the these options is specified, all of the distribution information is printed.
Table 5. rtshmdmp command options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| -l [bfFghHqrsS] | Prints information about data structures in shared memory. You can specify any number of the following options:  
|                | b Prints information about the base data structure.                                                                                 |
|                | f Prints information about flushers.                                                                                                      |
|                | F Prints information about data feeds.                                                                                                    |
|                | g Prints information about groups.                                                                                                         |
|                | h Prints information about hash buckets.                                                                                                   |
|                | H Prints information about feed handlers.                                                                                                 |
|                | q Prints information about query threads.                                                                                                  |
|                | r Prints information about record types.                                                                                                  |
|                | s Prints information about stores.                                                                                                         |
|                | S Prints information about storage systems.                                                                                               |
|                | t Prints information about transport-related data structures.                                                                             |
|                | If none of these options is specified, information about all of the data structures is printed.                                           |
| -s shared_memory_key | Specifies the shared memory key that was used when you started DSE with the rtloader command.                                        |
| -t [store_id [symbol_name]] | Prints the ticks for the specified symbol in the specified store.  
|                | • If the symbol name is not provided, this command prints all of the ticks for all of the symbols in the specified store.  
|                | Do not provide a symbol name for stores that use a simple schema.                                                                      |
|                | • If the store ID is not provided, this command prints all of the ticks for all of the symbols in all of the stores.                        |
| -v             | Prints version information.                                                                                                              |
Table 5. rtlshmdmp command options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• If the symbol name is not provided, this command prints the tick and metadata information of all of the symbols in the specified store.</td>
</tr>
<tr>
<td></td>
<td>Do not provide a symbol name for stores that use a simple schema.</td>
</tr>
<tr>
<td></td>
<td>• If the store ID is not provided, this command prints the tick and metadata information of all of the symbols in all of the stores.</td>
</tr>
<tr>
<td>--help</td>
<td>Displays help information for this command. This information is also displayed when the command is run without options or with incorrect options.</td>
</tr>
</tbody>
</table>

rtlstat utility

The rtlstat utility monitors the operation and performance of DSE.

The rtlstat command syntax is:

```
rtlstat -[bBdDehIlmpPStTwvz] [ -r [interval] ] [ -s shared_memory_key ] [ --help ]
```

The command options are summarized in Table 6. This section includes additional information about some of these options.

Table 6. rtlstat command options

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-b</td>
<td>Displays statistics for the feed handler buffer.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “-b option: Display statistics for the feed handler buffer” on page 45.</td>
</tr>
<tr>
<td>-B</td>
<td>Displays statistics for the backing store.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “-B option: Display statistics for the backing store” on page 45.</td>
</tr>
<tr>
<td>-d</td>
<td>Displays statistics for feed handlers.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “-d option: Display statistics for feed handlers” on page 45.</td>
</tr>
<tr>
<td>-D</td>
<td>Displays feed rate summary statistics.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “-D option: Display feed rate summary statistics” on page 46.</td>
</tr>
<tr>
<td>Options</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>-e</td>
<td>Displays the statistics for bad data that is discarded. For more information, see “-e option: Display statistics for discarded data” on page 46.</td>
</tr>
<tr>
<td>-f</td>
<td>Displays statistics for flushing data to the store. For more information, see “-f option: Display statistics for data flushes” on page 46.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays the current high availability status. For more information, see “-h option: Display the current high availability status” on page 46.</td>
</tr>
<tr>
<td>-i</td>
<td>Displays statistics for new symbol creation. For more information, see “-i option: Display statistics for new symbol creation” on page 47.</td>
</tr>
<tr>
<td>-l</td>
<td>Displays statistics for spinlocks. For more information, see “-l option: Display statistics for spinlocks” on page 47.</td>
</tr>
<tr>
<td>-m</td>
<td>Displays shared memory usage. For more information, see “-m option: Display shared memory usage” on page 47.</td>
</tr>
<tr>
<td>-p</td>
<td>Displays performance statistics. For more information, see “-p option: Display performance statistics” on page 47.</td>
</tr>
<tr>
<td>-P</td>
<td>Displays partitioning statistics. For more information, see “-P option: Display partitioning statistics” on page 47.</td>
</tr>
<tr>
<td>-S</td>
<td>Displays summary statistics. For more information, see “-S option: Display summary statistics” on page 48.</td>
</tr>
<tr>
<td>-t</td>
<td>Displays transmit statistics. For more information, see “-t option: Display transmit statistics” on page 48.</td>
</tr>
<tr>
<td>-T</td>
<td>Displays transmit rate summary statistics. For more information, see “-T option: Display transmit rate summary statistics” on page 48.</td>
</tr>
<tr>
<td>-w</td>
<td>Displays statistics for metadata writes for each store. For more information, see “-w option: Display statistics for metadata writes for each store” on page 48.</td>
</tr>
</tbody>
</table>
Options | Description
-------|------------------
-v | Displays version information for DSE.
-z | Zeros out statistics (for connection and lock counters) so that their accumulation can be monitored.
-r | Sets an interval, in seconds, at which to produce reports. The default is five seconds.
-s shared_memory_key | Specifies the shared memory key that was used when you started DSE with the rloader command. For more information, see “-s option: Specify a non-default shared memory key” on page 48.
--help | Displays help information for this command.

-b option: Display statistics for the feed handler buffer

The -b option shows how many ticks were loaded by each feed and how much of the feed’s free-list buffer is currently used.

-B option: Display statistics for the backing store

For each store, the -B option produces a report that shows the following information:

- The store status.
- The number of symbols in the store (for stores using a master-detail relationship only).
- The number of ticks in shared memory for the store.
- The number of ticks in shared memory for the store that have been flushed to the storage system.
- The number of attempts by DSE to retry failed storage system operations.
- The number of attempts by DSE to reconnect to a data store.

The following is a sample report:

<table>
<thead>
<tr>
<th>Store Number</th>
<th>Store Status</th>
<th>Symbol Count</th>
<th>Tick Count</th>
<th>Flush Count</th>
<th>Retry Count</th>
<th>Reconnect Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ONLINE</td>
<td>285</td>
<td>1000</td>
<td>962</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>ONLINE</td>
<td>180</td>
<td>900</td>
<td>762</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>TOTAL:</td>
<td></td>
<td>465</td>
<td>1900</td>
<td>1724</td>
<td>31</td>
<td>2</td>
</tr>
</tbody>
</table>

-d option: Display statistics for feed handlers

The -d option produces a report that shows the following information:

- How many times a feed handler tried to connect to its source
- How many messages and ticks the feed handler received
- How many bytes the received messages and ticks represent
- The source from which the feed handler is getting data
-D option: Display feed rate summary statistics

The -D option produces a report that shows the following information:
- The number of ticks per second that were received since DSE started
- The number of ticks per second in the last interval
- The total number of ticks that were received

-e option: Display statistics for discarded data

The -e option produces a report that shows the statistics for bad data that is discarded. For example:

<table>
<thead>
<tr>
<th>Flusher (store)</th>
<th>Last Discarded</th>
<th>Last Discard At</th>
<th>Last Flushed At</th>
<th>Total Discarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-</td>
<td>2007-03-28 01:23:45</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

TOTAL: 205

-f option: Display statistics for data flushes

The -f option produces a report that shows, for each fluser, the following information:
- The number of ticks flushed in the last flush
- How long the last flush took to complete
- The time of the last flush
- The total number of ticks flushed
- The total time spent flushing

For example:

<table>
<thead>
<tr>
<th>Flusher (store)</th>
<th>Last Flushed</th>
<th>Last Flushed At</th>
<th>Total Flushed</th>
<th>Duration (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE 1 (SIMPLE)</td>
<td>500</td>
<td>2007/07/31 14:14:04</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
<td>2007/07/31 14:14:04</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>STORE 2 (MASTER-DETAIL)</td>
<td>606</td>
<td>2007/07/31 14:14:04</td>
<td>606</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>394</td>
<td>2007/07/31 14:14:04</td>
<td>394</td>
<td>0</td>
</tr>
</tbody>
</table>

TOTAL: 2000

-h option: Display the current high availability status

The -h option produces a report that shows the state of the DSE instance (ACTIVE, STANDBY, STARTED, or SYNCHRONIZING). The exit status of this command is the same as the return value of the rtl_state_get() API function.

If the DSE instance is in SYNCHRONIZING state, the report shows the following information:
- The host name of the live DSE instance
- The stage of the synchronization
- The percentage of stores and symbols synchronized in that stage
-i option: Display statistics for new symbol creation

The -i option produces a report that shows:
- How many symbols are in each store
- How many symbols were inserted
- How many symbols are waiting to be inserted

Stores that use a simple schema are not included in these statistics.

-l option: Display statistics for spinlocks

The -l option displays a report that shows the following information:
- How many spinlocks were obtained to perform various operations
- How often these spinlocks failed

DSE uses spinlocks for concurrency control. Because each failure causes the process to perform a busy wait, a cumulative total of number of spins and the maximum spin is reported.

-m option: Display shared memory usage

The -m option displays a report that shows:
- How many ticks were received during the current run
- How many ticks are currently available on the free list
- How many ticks are in shared memory

This option also displays the percent utilization of both the security and tick pools.

-p option: Display performance statistics

The -p option displays a set of performance statistics that includes information about the efficiency of loading, flushing, and purging.

-P option: Display partitioning statistics

The -P option displays, for each store and feed that writes to the store, the following statistics:
- The total number of ticks that have been partitioned since the current DSE session started.
- The number of ticks that are currently being partitioned.
- The number of ticks that are waiting to be partitioned.
- The number of ticks that were misrouted.

The report also displays totals of each of these statistics at the store and system level.

For example:

<table>
<thead>
<tr>
<th>Store Number</th>
<th>Feed Number</th>
<th>Partitioned Count</th>
<th>Current Count</th>
<th>Pending Count</th>
<th>Mispartitioned Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>200000</td>
<td>0</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>300000</td>
<td>2000</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>
**-s option: Specify a non-default shared memory key**

Use the -s option to specify the shared memory key that was used when you started DSE.

The order of precedence for specifying the shared memory key is as follows, from highest to lowest:
1. Command line (using the -s option)
2. RTLSHMKEY environment variable
3. Default value (268 658 176)

**-S option: Display summary statistics**

The -S option produces a report that shows summary statistics about:
- Current loading efficiency
- Loading efficiency since DSE started

**-t option: Display transmit statistics**

The -t option produces a report with a line that summarizes data for each feed. The report shows the following information:
- How many times a feed handler tried to connect to its target
- How many messages and ticks the feed handler sent
- How many bytes the sent messages and ticks represent
- The target to which the feed handler is sending data

**-T option: Display transmit rate summary statistics**

The -T option produces a report that shows the following information:
- The number of messages that were transmitted per second to a target since DSE started
- The number of ticks per second that were transmitted to a target since the last report
- The total number of ticks that were transmitted

**-w option: Display statistics for metadata writes for each store**

The -w option produces a report that shows the following information:

<table>
<thead>
<tr>
<th>Store No.</th>
<th>Type</th>
<th>Last Run Start</th>
<th>Last Run Time</th>
<th>Symbols Duration</th>
<th>Symbols Updated</th>
<th>Symbols Aborted</th>
<th>Symbols Locked</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>Auto</td>
<td>yyyy/mm/dd hh:mm:ss</td>
<td>nnnn secs</td>
<td>nnnnnnnn</td>
<td>nnnnnn</td>
<td>nnnnnn</td>
<td></td>
</tr>
<tr>
<td>[3] TOTAL:</td>
<td></td>
<td>yyyy/mm/dd hh:mm:ss</td>
<td>nnnn secs</td>
<td>nnnnnn</td>
<td>nnnnnn</td>
<td>nnnnnn</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store No.</th>
<th>Type</th>
<th>Min. run</th>
<th>Max. run</th>
<th>Avg. run</th>
<th>Runs</th>
<th>Runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>Auto</td>
<td>nnnn secs</td>
<td>nnnn secs</td>
<td>nnnn secs</td>
<td>nnnn secs</td>
<td>nnnn</td>
</tr>
</tbody>
</table>

---

2 3 100000 3000 500 0

TOTAL: 600000 5000 3600 0

2 3 100000 3000 500 0

TOTAL: 600000 5000 3600 0

- S option: Display summary statistics

The -S option produces a report that shows summary statistics about:

- Current loading efficiency
- Loading efficiency since DSE started

-t option: Display transmit statistics

The -t option produces a report with a line that summarizes data for each feed. The report shows the following information:

- How many times a feed handler tried to connect to its target
- How many messages and ticks the feed handler sent
- How many bytes the sent messages and ticks represent
- The target to which the feed handler is sending data

-T option: Display transmit rate summary statistics

The -T option produces a report that shows the following information:

- The number of messages that were transmitted per second to a target since DSE started
- The number of ticks per second that were transmitted to a target since the last report
- The total number of ticks that were transmitted

-w option: Display statistics for metadata writes for each store

The -w option produces a report that shows the following information:

<table>
<thead>
<tr>
<th>Store No.</th>
<th>Type</th>
<th>Last Run Start</th>
<th>Last Run Time</th>
<th>Symbols Duration</th>
<th>Symbols Updated</th>
<th>Symbols Aborted</th>
<th>Symbols Locked</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>Auto</td>
<td>yyyy/mm/dd hh:mm:ss</td>
<td>nnnn secs</td>
<td>nnnnnnnn</td>
<td>nnnnnn</td>
<td>nnnnnn</td>
<td></td>
</tr>
<tr>
<td>[3] TOTAL:</td>
<td></td>
<td>yyyy/mm/dd hh:mm:ss</td>
<td>nnnn secs</td>
<td>nnnnnn</td>
<td>nnnnnn</td>
<td>nnnnnn</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store No.</th>
<th>Type</th>
<th>Min. run</th>
<th>Max. run</th>
<th>Avg. run</th>
<th>Runs</th>
<th>Runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>Auto</td>
<td>nnnn secs</td>
<td>nnnn secs</td>
<td>nnnn secs</td>
<td>nnnn secs</td>
<td>nnnn</td>
</tr>
</tbody>
</table>
==========================================================================

- Lines 1 and 2 show the start time, duration, and the number of symbols that
  were updated in the last automatic and manual metadata writes for each store.
  Three columns provide specific information about the symbols:

  **Symbols updated**
  The number of symbols that were successfully updated.

  **Symbols aborted**
  The number of symbols for which updates were attempted, but failed.

  **Symbols locked**
  The number of symbols that were not updated because they were locked
  at the time of the metadata write.

If a metadata write is in progress, the "Run duration" column shows "[in prog]."

- Line 3 shows the total number of symbols that were updated, aborted, or locked
  for all stores and all metadata writes.

- Lines 4 and 5 show the minimum, maximum, and average durations for all
  completed metadata writes for all stores. For manual metadata writes, these lines
  also show the number of pending writes (due to rtlmode -w requests).

- Line 6 shows the total number of completed metadata writes for all stores.

Stores that use a simple schema are not included in these statistics.

---

**rtl_make_pass utility**

The rtl_make_pass utility creates encrypted passwords for use in the configuration
file.

The rtl_make_pass command syntax is:

```bash
rtl_make_pass [ -v ] name
```

where:

- `-v` Provides the DSE version information and exits the utility.

`name` Is the name of the group or storage system user for which you are creating
a password.

**Example:**

```
% rtl_make_pass rtltest
Password for rtltest: ********** (Enter "marshmello"; it will not be echoed)
Retype Password for rtltest: **********
encrypted pass: c9cf29ebcc8c2e67c7a7b797c7463
```

In this example, the corresponding parameters in the configuration file that use
these values are:

```plaintext
GROUP01_NAME "rtltest"
GROUP01_PASSWORD "c9cf29ebcc8c2e67c7a7b797c7463"
```

Two parameters in the configuration file require an encrypted password:

- `GROUPXX_PASSWORD`
- `STORESYSXX_PASSWORD`
Chapter 9. System configuration parameters

The DSE configuration file contains configuration parameters that describe your system and the data that you are reading. DSE provides sample configuration files.

These sample configuration files are in the /config subdirectories of the simple, generic, ha and restart examples. You can edit these files, or create your own configuration file. See Chapter 5, “DSE directory structure,” on page 13 for the location of these files.

To create your own configuration file:
1. Select a sample configuration file to use as the basis for your own configuration file.
2. Customize the configuration file with the parameter settings that you need for your system and the data that you are reading. An IBM consultant can help you set up the initial values in this file.
3. When the production version of this file is ready, you can keep it in its current location or put it anywhere on your system.

Important: Enclose parameter values that are strings in quotation marks. Do not enclose numerical parameter values in quotation marks.

The following table shows the types of system configuration parameters.

<table>
<thead>
<tr>
<th>Type of system configuration parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“General system configuration parameters” on page 52</td>
<td>Defines some basic settings for your system.</td>
</tr>
<tr>
<td>“Query server configuration parameters” on page 60</td>
<td>Sets up your query server configuration.</td>
</tr>
<tr>
<td>“Shared memory configuration parameters” on page 62</td>
<td>Defines the shared memory area that DSE can use and defines how DSE purges data.</td>
</tr>
<tr>
<td>“Feed configuration parameters” on page 54</td>
<td>Defines the feeds from which DSE gets data.</td>
</tr>
<tr>
<td>“Transport configuration parameters” on page 82</td>
<td>Specifies various values for the transport layer.</td>
</tr>
<tr>
<td>“Store configuration parameters” on page 74</td>
<td>Customizes how DSE stores data.</td>
</tr>
<tr>
<td>“Storage system configuration parameters” on page 71</td>
<td>Configures the storage system that you are using with DSE.</td>
</tr>
<tr>
<td>“Group configuration parameters” on page 57</td>
<td>Configures group access to data. DSE controls access to data by using group-level parameters.</td>
</tr>
<tr>
<td>“Record configuration parameters” on page 61</td>
<td>Configures the records for each of the feeds from which DSE is getting data.</td>
</tr>
<tr>
<td>“Feed handler configuration parameters” on page 56</td>
<td>Configures the feed handlers that DSE uses to control the data streams.</td>
</tr>
<tr>
<td>“High availability configuration parameters” on page 57</td>
<td>Configures a high availability system that consists of multiple DSE instances.</td>
</tr>
<tr>
<td>“Trace configuration parameters” on page 79</td>
<td>Configures the trace logging activity for diagnosing problems with DSE.</td>
</tr>
</tbody>
</table>
Related reference
Chapter 11, “Rules for delimited identifiers,” on page 85

The DSE configuration file can contain delimited identifiers for some of the parameters. Follow these rules and guidelines when using delimited identifiers in the configuration file.

General system configuration parameters

Use the general system configuration parameters to define some basic settings for your system.

**ENABLE_NEWSYM**

This parameter specifies whether DSE accepts new symbols from the data stream.

Valid parameter values are:

1  DSE accepts new symbols as they appear in the data stream.
0  DSE ignores data for any new symbols.

Additional control of new symbols is available with the STOREXX_NEWSYMBOLS parameter. This parameter has a default value of 1.

**INCLUDE**

This parameter specifies an additional configuration file to include. For example, in addition to the main rtlconfig file, you can have a separate configuration file to define your record structure.

INCLUDE is either an absolute path or a path that is relative to the location of the rtlconfig file. DSE includes the text from the included file precisely where the INCLUDE variable is found.

This parameter has a default value of null.

**NODISKWRITE**

This parameter specifies whether DSE flushes ticks to permanent storage for any of the data stores.

Valid parameter values are:

0  DSE flushes ticks to permanent storage. This is the default value.
1  DSE does not write ticks to permanent storage during flushing.

Note that while DSE does not write ticks to permanent storage when NODISKWRITE=1, these ticks are included in the statistics for data flushes (from the rtlstat -f command).

You can also disable writes to permanent storage with the STOREXX_NODISKWRITE parameter.

**RESTART_ENABLE**

This parameter specifies whether DSE operates in restartable mode or non-restartable mode.
Valid parameter values are:

0  DSE runs in normal, non-restartable mode. During an unplanned outage or system crash, unflushed ticks data in shared memory can be lost.

1  DSE runs in restartable mode. During an unplanned outage or system crash, tick records are recoverable (symbols data and metadata are not recoverable). This mode is supported only for transactional feeds that use WebSphere MQ for AIX, Linux, and Solaris, Version 6.0 as a transaction manager, configured for two-phase commit with DB2. For more information about this functionality and its restrictions, see Chapter 12, “Restartable mode for high availability,” on page 87.

This parameter has a default value of 0.

**RTL_LOG_LOCALTIME**

This parameter specifies the time zone that DSE uses in the log file.

Valid parameter values are:

1  DSE uses local time.

0  DSE uses Coordinated Universal Time (UTC).

RTL_LOG_LOCALTIME has a default value of 1.

**RTL_LOG_PATH**

This parameter specifies the path of the message log file. The path can be either an absolute path or a path that is relative to the location of the rtlconfig file.

This value can be ignored by invoking DSE with the -n option. If the -n option is used, the log messages are sent to stderr.

*Example:* The following fragment of the rtlconfig file defines the path of the log file as "rtl.log".

```
#---------------------------------------------------------------
# Logging and message file info
#---------------------------------------------------------------
RTL_LOG_PATH  "rtl.log"  #Path of log file for messages
```

This parameter is a string with a default value of "rtl.log".

**RTL_SYS_MSG_PATH**

This parameter specifies the path of the message file for system messages.

The value of RTL_SYS_MSG_PATH parameter can be one of the following:

- An absolute path
- A path that is relative to the directory that contains the DSE configuration file
- A file that is located in the DSE_INSTALL_DIR/msg directory

The rtlsysmsgs.txt message file contains all of the messages that can be displayed by the core DSE server. When an error occurs, DSE writes the message to the standard error output or to the log file, if the log file is specified by the RTL_LOG_PATH parameter in the rtlconfig file.

This parameter is a string with a default value of "rtlsysmsgs.txt".
UDPX

This parameter specifies a user-defined configuration parameter. The ten user-defined parameter variables are UDP0 through UDP9, all of which are globally visible. UDPX can be a string or an integer.

DSE assumes that user-defined parameters that are enclosed in quotation marks are strings, and that user-defined parameters that are not enclosed in quotation marks are integers.

To access the values of user-defined variables, use the rtl_get_udp_value() API function that is described in Chapter 27, “DSE API reference,” on page 285.

Feed configuration parameters

DSE is designed to process input from multiple feeds of data. For each feed, you can set these parameters to customize loading.

The XX in these parameter names is the feed number, and can be 00 to 99.

FEEDXX_ACCEPT_TICKS_ALWAYS

This parameter specifies whether DSE always accepts ticks and adds them to shared memory, or only when the data store is online.

Valid parameter values are:

0  DSE accepts ticks and adds them to shared memory only when the data store is online. DSE discards all ticks when the store is offline.
1  DSE accepts ticks and adds them to shared memory regardless of whether the data store is online or offline. When an offline data store is brought back online, the ticks from shared memory can be written to the data store (unless the ticks are purged from shared memory because of memory constraints).

This parameter has a default value of 1.

See Chapter 19, “Enabling error recovery for storage systems,” on page 131 for a description of how a data store goes offline and online.

FEEDXX_GAPFILL

This parameter specifies whether the feed is real time or gap fill.

Valid parameter values are:

0  The feed is real time.
1  The feed is gap fill.

This parameter has a default value of 0.

FEEDXX_HANDLER_NAME

This parameter specifies the name of the feed handler.

FEEDXX_HANDLER_NAME is a string that is the same as the value of HANDLERXX_NAME of the corresponding feed handler.
This parameter is required.

**FEEDXX_INFO**

This parameter specifies additional information about the feed that is interpreted by the feed handler.

This information can be accessed by calling the `rtl_feed_get_info` API function that is described in Chapter 27, “DSE API reference,” on page 285.

`FEEDXX_INFO` is a string with a default value of `null`.

**FEEDXX_TRANSPORT_NAME**

This parameter specifies the name of the transport associated with this feed.

The value of `FEEDXX_TRANSPORT_NAME` must match the name of the transport as specified in `TRANSPORTXX_NAME`.

This parameter is required.

**FEEDXX_SOURCE**

This parameter specifies the initial data source for this feed.

The format of the source name is defined by the transport. See “Source and target names for pre-built transports” on page 171 for information on the source name format for each type of pre-built transport.

This parameter is required.

**FEEDXX_STORE_NAME**

This optional parameter specifies the name of the storage location.

**Note:** This parameter is required by the sample feed handlers that DSE provides.

The `FEEDXX_STORE_NAME` parameter value is a string. It should match one of the values of the `STOREXX_NAME` parameters. For example:

```
FEED01_STORE_NAME "abc"
STORE09_NAME "abc"
```

If you want to use special characters, spaces, or mixed case letters in a store name, you must follow the rules for delimited identifiers. See Chapter 11, “Rules for delimited identifiers,” on page 85.

This parameter has a default value of `null`.

**FEEDXX_TARGET**

This parameter specifies the data target for this feed.

The format of the value of `FEEDXX_TARGET` is dependent on the target to which DSE writes. See “Source and target names for pre-built transports” on page 171 for information on the target name format for each type of pre-built transport.

This parameter has a default value of `null`. 

Chapter 9. System configuration parameters 55
**FEEDXX_WRITECANBLOCK**

This parameter specifies whether blocking is permitted when the feed handler is writing to shared memory.

The feed handler blocks when the percentage of shared memory in use is larger than the value of PURGE_HWM. Some data sources require that the feed handler read data within a certain period of time, otherwise overruns can occur.

Valid parameter values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DSE does not allow blocking, which results in an error message and causes DSE to shut down if blocking is required.</td>
</tr>
<tr>
<td>1</td>
<td>DSE allows blocking while writing.</td>
</tr>
</tbody>
</table>

This parameter has a default value of 0.

---

**Feed handler configuration parameters**

DSE uses feed handlers to process the data streams. You can set these parameters to configure the feed handler.

The XX in these parameter names is the feed handler number, and can be 00 to 99.

**HANDLERXX_INFO**

This parameter specifies additional information that is interpreted by the feed handler.

This information can be accessed by calling the rtl_handler_get_info API function that is described in [Chapter 27, “DSE API reference,” on page 285](#).

HANDLERXX_INFO is a string with a default value of null.

**HANDLERXX_LIB_PATH**

This parameter specifies the path of the shared library for the feed handler.

The value of HANDLERXX_LIB_PATH is a string and can be one of the following:

- An absolute path
- A file that is located in the DSE_INSTALL_DIR/lib directory
- A file that is located in a directory in the LD_LIBRARY_PATH (or LIBPATH for AIX) environment variable

**Recommendation:** Specify an absolute path if DSE is part of a configuration in which DSE is started automatically. For example, DSE can be started automatically on system startup, by a cron job, or in a high availability configuration.

This parameter is required.

**HANDLERXX_MSG_PATH**

This parameter specifies the path of the message file for the feed handler.

The value of HANDLERXX_MSG_PATH can be one of the following:

- An absolute path
- A path that is relative to the directory that contains the DSE configuration file
A file that is located in the DSE_INSTALL_DIR/msg directory

This parameter is required.

**HANDLERXX_NAME**

This parameter specifies the name of the feed handler.

HANDLERXX_NAME is a text string. This parameter is required.

---

**Group configuration parameters**

DSE controls access to data by using group-level permissions. You can set these parameters to configure the group access.

The XX in these parameter names is the group number, and can be 00 to 99.

**GROUPXX_AUTHLEVEL**

This parameter specifies the authorization level of the group.

To access a store, the group authorization level must be greater than or equal to the store authorization level.

This parameter is an integer with a maximum value of 2 147 483 647. This parameter has a default value of 1.

**GROUPXX_NAME**

This parameter specifies the name of the group.

GROUPXX_NAME is a text field that corresponds to the name that is used for client connection requests. Clients that use the wrapper or client API calls to query DSE require a corresponding setting to reach the server.

This parameter has a default value of null.

**GROUPXX_PASSWORD**

This parameter specifies the password for the group.

GROUPXX_PASSWORD is an encrypted string that corresponds to the unencrypted password that is used for client-connection-request API calls. You can use the rtl_make_pass utility to encrypt a password. Clients that use the wrapper or client API calls to query DSE require a corresponding setting to reach the server.

This parameter has a default value of null.

---

**High availability configuration parameters**

Use these parameters to configure a high availability system that consists of multiple DSE instances.

**Related concepts**

[Chapter 12, “Restartable mode for high availability,” on page 87](#)

DSE can be run in restartable or non-restartable mode. In restartable mode, DSE guards against data loss in case of system failure.
A high availability cluster, consisting of multiple host systems and peer DSE instances, ensures continuous availability of data if a failure occurs.

**HA_COMM_TIMEOUT**

This parameter specifies the maximum length of time, in seconds, that a DSE instance waits for a call to communicate with its peer DSE instances to complete.

HA_COMM_TIMEOUT is a positive integer and has a default value of 30.

**HA_ENABLE**

This parameter enables high availability support.

Valid parameter values are:

0  High availability is not enabled.
   If high availability is not enabled, all high availability configuration parameters are ignored.

1  High availability is enabled.
   If high availability is enabled, the high availability communications library API is loaded. When this library is loaded, the rtl_ha_init function is called, which allocates and initializes the high availability user data area and a data structure for communicating with each of the peer DSE instances.

This parameter has a default value of 0.

**HA_INFO**

This parameter specifies user information for the high availability communication library.

The library can access the value of HA_INFO by using API functions. HA_INFO is a string with a default value of null.

**HA_LIB_PATH**

This parameter specifies the path to the high availability communication library that you use.

The value of HA_LIB_PATH can be one of the following:

- An absolute path
- A file that is located in the DSE_INSTALL_DIR/lib directory
- A file that is located in a directory in the LD_LIBRARY_PATH (or LIBPATH for AIX) environment variable

HA_LIB_PATH is a string with a default value of null.

**HA_MAXMSG_SIZE**

This parameter specifies the maximum number of bytes that can be sent in a message between DSE peer instances in a high availability cluster.

HA_MAXMSG_SIZE is a positive integer and has a default value of 15000.
HA_MSG_PATH

This parameter specifies the path of the message file for high availability communication library-specific messages.

The value of HA_MSG_PATH can be one of the following:
- An absolute path
- A path that is relative to the directory that contains the DSE configuration file
- A file that is located in the DSE_INSTALL_DIR/msg directory

HA_MSG_PATH is a string with a default value of null.

HA_RETRY_DELAY

This parameter specifies the length of time, in seconds, that a standby DSE instance waits after losing a connection, before it tries to reconnect to the live DSE instance.

HA_RETRY_DELAY is a positive integer and has a default value of 0.

HA_SAVEBUF_SIZE

This parameter specifies the size, in bytes, of the buffer for saving messages and ticks while metadata is retrieved.

HA_SAVEBUF_SIZE is a positive integer and has a default value of 50 000.

HA_SHARED_DB

This parameter specifies whether the DSE instances in a high availability cluster share the same storage systems.

Valid parameter values are:
0   Each DSE instance is connected to different set of storage systems, and each DSE instance is solely responsible for updating its own set of storage systems.
1   All of the DSE instances are connected to the same set of storage systems, and each DSE instance updates the same set of storage systems.

This parameter has a default value of 0.

HA_STANDBY_API

This parameter specifies whether a C-API or Java API call can connect to a DSE instance that is in STANDBY state.

Valid parameter values are:
0   The API call to connect to the DSE instance fails.
1   The API call to connect to the DSE instance is successful.

This parameter has a default value of 0.

Note that HA_STANDBY_API applies only when the client connects to a specific store. If the client wants to connect to a DSE instance without connecting to a specific store (for example, to retrieve system statistics), it can connect using the rtl_attach API function.
**HA_STANDBY_PUB**
This parameter specifies whether the ticks are published to feed targets when the DSE instance is in STANDBY state.

Valid parameter values are:
0 The ticks are not published to the feed targets.
1 The ticks are published to the feed targets.

This parameter has a default value of 0.

**HA_SYNC_DELAY**
This parameter specifies the length of time, in seconds, that a standby DSE instance waits before it starts to synchronize with a live DSE instance.

Specify a delay if a problem with synchronization is evident.

HA_SYNC_DELAY is a positive integer and has a default value of 0.

**HAXX_PEER_INFO**
This parameter specifies user information about a peer DSE instance.

The high availability communication library defines the contents of the string and uses this information to establish a connection to the peer DSE instance. Define one HAXX_PEER_INFO entry for each peer DSE instance in the high availability cluster. XX is the identifier for the DSE instance, and it can have a value of 00 through 09.

Example:

For the high availability communication library, this string specifies a fully qualified host name and port number separated by a colon. If you use port 5999 for peer communication, and the fully qualified hostname of the peer DSE instance is host1.example.org, then the information string for the peer instance is host1.example.org:5999.

HAXX_PEER_INFO is a string with a default value of null.

**HAXX_PEER_NAME**
This parameter specifies the name of a peer DSE instance.

During synchronization, you can display the name of the peer instance using the rtlstat -h command. XX is the identifier for the DSE instance, and it can have a value of 00 through 09.

HAXX_PEER_NAME is a string with a default value of null.

---

**Query server configuration parameters**

Use the query server configuration parameters to set up your query server configuration.
**MAX_QUERY_CONNECTS**
This parameter specifies the maximum number of query connections that can be made to DSE simultaneously.

The number of query connections is equivalent to the number of users. This value must be a non-negative integer that is greater than or equal to the value of QUERY_THREADS.

This parameter has a default value of 5.

**QUERY_THREADS**
This parameter specifies the initial number of query threads that are started to handle query requests from clients (such as the wrapper) that use the C or Java API.

Each query thread uses a socket for internal communication, starting with the first port after the one that is defined in RTL_SERVICE_NAME. The value is incremented by one for each additional thread.

**Example:** If the RTL_SERVICE_NAME maps to port 12345 and QUERY_THREADS is set to 3, the query threads use ports 12346, 12347, and 12348 internally.

This parameter has a default value of 3. The maximum number of query threads that can be configured is 100.

**RTL_SERVICE_NAME**
This parameter specifies the port from which DSE receives query requests.

You can specify either a name or a port number. If the value contains only digits, DSE assumes that the value is a port number. If the value contains other characters, DSE assumes that the value is a service entry that is defined in the /etc/services file. Clients that use a wrapper or client API calls to query DSE require a corresponding setting to reach the server.

This parameter is a string with a default value of "db2_rtl".

---

**Record configuration parameters**
DSE uses records to define the structure and characteristics of ticks and symbol metadata in shared memory. You can configure the records with these parameters.

The XX in these parameter names is the record number, and can be 00 to 99.

**RECORDXX_NAME**
This parameter specifies the name of the record.

RECORDXX_NAME is a text field. This parameter is required.

**RECORDXX_META_SCHEMA**
This parameter specifies the structure of the symbol metadata in DSE memory.

RECORDXX_META_SCHEMA is a string with a default value of null.
Note: This parameter is ignored for stores that use a simple schema.

The format for RECORDXX_META_SCHEMA is as follows:

\[(field_name \ type \ [NOT \ NULL] \ [WITH \ DEFAULT \ def_val] \ [STORE \ [READ | READWRITE \ [AUTO]] \ [store_col_name]], \ldots)\]

The RECORDXX_META_SCHEMA variables and keywords are as follows:

**field_name**
Name of the metadata field in shared memory. field_name consists of one to 32 alphanumeric characters. The first character must be alphabetic.

If you want to use special characters, spaces, or mixed case letters in a field name, you must follow the rules for delimited identifiers. See Chapter 11, “Rules for delimited identifiers,” on page 85.

**type** Type of the field. Allowable types are:

- bigint
- boolean
  This type supports fields of boolean types. The default value for a boolean field must be either true or false.
- byte \( (n) \)
  This type supports fields of byte array types of length \( n \). The default value for a byte field must be specified as a hexadecimal string in single quotation marks, and the length of the string can be up to 64 characters. For example, ‘13579ace’. This maximum length corresponds to the maximum of 32 bytes for the default in binary.
- char \( (n) \)
  If type is char, you must provide a numeric size \( n \) enclosed in parentheses, where \( n \) is the length of the string.
- date
- double
- float \( (n) \)
  If type is float, you can provide an optional precision value \( n \) enclosed in parentheses. Based on the value of \( n \), DSE maps the field to a corresponding DSE type:

<table>
<thead>
<tr>
<th>( n ) value in float ( (n) )</th>
<th>DSE type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not specified</td>
<td>double</td>
</tr>
<tr>
<td>Between 1 and 24</td>
<td>real</td>
</tr>
<tr>
<td>Between 25 and 52</td>
<td>double</td>
</tr>
</tbody>
</table>

- int
- real
- smallint
- time_t
- timestamp
- timestamp(\( \text{rtl}\_\text{datetime} \)) [efficient 8-byte timestamp with a precision of tens of microseconds]
- timestamp(\( \text{rtl}\_\text{java}\_\text{date} \)) [milliseconds since the epoch (1 January 1970)]
- timestamp(time_t) [seconds since the epoch (1 January 1970)]
• varchar (n)
  If type is varchar, you must provide a numeric size n enclosed in parentheses, where
  n is the length of the string.

Note: Types time_t and timestamp are deprecated. Use
  timestamp(rtl_datetime) and timestamp(time_t) instead.

NOT NULL
Optional. This field can never be null.

WITH DEFAULT def_val
Optional. If the field is null, def_val is used.

Restrictions:
• The only allowable default value for a date type is current date.
• The only allowable default value for a timestamp type is current
  timestamp.
• The maximum length of a default value for char and varchar types is 32
  characters.
• The default value for a byte field must be specified as a hexadecimal
  string in single quotation marks. The length of the string can be up to 64
  characters. For example: '13579ace'. This maximum length corresponds
  to the maximum of 32 bytes for the default in binary.

STORE READ
Optional. If the STOREXX_META_MODE is not 0, the indicated field of
  metadata is read from the data store into shared memory when DSE starts.

For more information, see the STOREXX_META_MODE parameter
“STOREXX_META_MODE” on page 75.

STORE READWRITE
Optional. If the STOREXX_META_MODE is not 0, the indicated field of
  metadata is written to the data store when the rtlmode -w command is run
  or when a new symbol is added. The field cannot be written to the data
  store automatically.

For more information, see the STOREXX_META_MODE parameter
“STOREXX_META_MODE” on page 75.

STORE READWRITE AUTO
Optional. The indicated field of metadata can be written to the data store:
• Automatically, if STOREXX_META_MODE is 2.
• Manually, if the STOREXX_META_MODE is not 0, with the rtlmode -w
  command.

When a new symbol is added, the metadata is written to the store
  automatically.

For more information, see the STOREXX_META_MODE parameter
“STOREXX_META_MODE” on page 75.

store_col_name
Optional. Name of the metadata column in the data store. store_col_name is
  allowed only if you use the STORE keyword. Specify store_col_name if the
  name of the column differs from the name of the field in shared memory.

An example of a RECORDXX_META_SCHEMA parameter value is:
RECORD66_META_SCHEMA "(
  last_trade_timestamp timestamp, \n  last_trade_price double store read last_price, \n  last_trade_volume int store read last_volume, \n  high_price double with default -1 store readwrite auto, \n  low_price double with default -1 store readwrite auto, \n  cum_volume bigint with default 0 store readwrite, \n  num_trades int with default 0 store readwrite, \n  last_sequence_no int with default -1 not null \n)"

In this example, DSE handles the metadata fields as follows:
- last_trade_timestamp is not read from the data store and exists only in shared memory.
- last_trade_price and last_trade_volume are read from the data store and are stored in shared memory. These fields are not written to the data store. The names of the columns in the data store differ from the names of the fields in shared memory.
- high_price and low_price are written to the data store automatically, but only if STOREXX_META_MODE is 2.
- cum_volume and num_trades are read from the data store, stored in shared memory, and are written to the data store only when the `rtlmode -w` command is run. These fields are both defined with a default value of 0.
- last_sequence_no is not read from the data store and exists only in shared memory. This field is defined with a default value of -1 and cannot be null.

**Related concepts**

[Chapter 10, “Supported data types,” on page 83](#)

DSE provides or interacts with different data type systems that developers can use when working with different kinds of data streams and storage systems.

**RECORDXX_META_SEQUENCE**

This parameter specifies the field or fields of the symbol metadata structure that define a unique sequential identification number.

This identification number is used to synchronize symbol metadata between two DSE instances in a high availability DSE configuration. RECORDXX_META_SEQUENCE is a string with a default value of null.

**Note:** This parameter is ignored for stores that use a simple schema.

The format for RECORDXX_META_SEQUENCE is as follows:

```
(fiel_name, ...) 
```

where `field_name` is the name of a field from the RECORDXX_META_SCHEMA definition that corresponds to this RECORDxx_META_SEQUENCE definition.

For example:

```
RECORD01_META_SCHEMA "(field1 smallint not null default -1, \n  field2 double default 0.0, \n  field3 integer not null default 0, \n  field4 float default -1)"
RECORD01_META_SEQUENCE "(field3, field1)"
```

In this example, the unique identification number has two components, field3 and field1. When comparing the metadata of two symbols to determine their order in the sequence, DSE first compares the values of field3 for both symbols. If they are
different, the symbol with the lesser of the two values comes first in the sequence. If the values for field3 are the same, DSE compares the values of field1.

You can include as many fields from the RECORDXX_META_SCHEMA definition in a RECORDxx_META_SEQUENCE definition as you want, with the following restrictions and requirements:

- Fields can only occur once. For example, "(field1, field1)" is not allowed.
- Each field must have both a NOT NULL clause and a WITH DEFAULT clause.
- The default value in each field’s WITH DEFAULT clause should be numerically lower than any expected value for the field. For example, if values for field1 start with 0 and increase numerically, set the default to -1. If the field is one of the text types (char or varchar), choose a default string whose collation order is lower than any expected string value.
- You must define a RECORDXX_METASEQUENCE for each RECORDXX_META_SCHEMA definition if high availability is enabled (HA_ENABLE = 1).

**RECORDXX_SCHEMA**

This parameter specifies the structure of ticks in DSE memory.

The value of RECORDXX_SCHEMA looks similar to an SQL statement describing the rows of a table. The format for RECORDXX_SCHEMA is as follows:

\[(field\_name\ type [NOT NULL] [WITH DEFAULT def_val], ...)

The RECORDXX_SCHEMA variables are as follows:

**field\_name**

Name of the field. field\_name consists of one to 32 alphanumeric characters. The first character must be alphabetic.

If you want to use special characters, spaces, or mixed case letters in a field name, you must follow the rules for delimited identifiers. See Chapter 11, “Rules for delimited identifiers,” on page 85.

**type**

Type of the field. Allowable types are:

- bigint
- boolean
  
  This type supports fields of boolean types. The default value for a boolean field must be either true or false.
- byte \((n)\)
  
  This type supports fields of byte array types of length \(n\). The default value for a byte field must be specified as a hexadecimal string in single quotation marks, and the length of the string can be up to 64 characters. For example, ‘13579ace’. This maximum length corresponds to the maximum of 32 bytes for the default in binary.
- char \((n)\)
  
  If type is char, you must provide a numeric size \(n\) enclosed in parentheses, where \(n\) is the length of the string.
- date
- double
- float \((n)\)
  
  If type is float, you can provide an optional precision value \(n\) enclosed in parentheses. Based on the value of \(n\), DSE maps the field to a
corresponding DSE type:

<table>
<thead>
<tr>
<th>$n$ value in float ($n$)</th>
<th>DSE type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not specified</td>
<td>double</td>
</tr>
<tr>
<td>Between 1 and 24</td>
<td>real</td>
</tr>
<tr>
<td>Between 25 and 52</td>
<td>double</td>
</tr>
</tbody>
</table>

- int
- real
- smallint
- time_t
- timestamp
- timestamp(rlt_datetime) [efficient 8-byte timestamp with a precision of tens of microseconds]
- timestamp(rlt_java_date) [milliseconds since the epoch (1 January 1970)]
- timestamp(time_t) [seconds since the epoch (1 January 1970)]
- varchar ($n$)

If type is varchar, you must provide a numeric size $n$ enclosed in parentheses, where $n$ is the length of the string.

**Note:** Types time_t and timestamp are deprecated. Use timestamp(rlt_datetime) and timestamp(time_t) instead.

**NOT NULL**
Optional. This field can never be null.

**WITH DEFAULT** def_val
Optional. If the field is null, def_val is used.

**Restrictions:**
- The only allowable default value for a date type is current date.
- The only allowable default value for a timestamp type is current timestamp.
- The maximum length of a default value for char and varchar types is 32 characters.
- The default value for a byte field must be specified as a hexadecimal string in single quotation marks. The length of the string can be up to 64 characters. For example: '13579ace'. This maximum length corresponds to the maximum of 32 bytes for the default in binary.

An example of a record schema is:
```
RECORD01_SCHEMA "(tstamp timestamp,
    TRDPRC_1 double with default -1,
    IRGCOND char (4) with default '')"
```

**Restriction:** For stores using a master-detail relationship, the first field of RECORDXX_SCHEMA must be of the type timestamp(rlt_datetime).
RECORDXX_SCHEMA is a text string.

This parameter is required.

**Related concepts**
**RECORDXX_SEQUENCE**

This parameter specifies the field or fields of a tick structure that define a unique sequential identification number.

This identification number is used to synchronize tick data between two DSE instances in a high availability DSE configuration. RECORDXX_SEQUENCE is a string with a default value of null.

The format for RECORDXX_SEQUENCE is as follows:

\[ \text{field}_1, \ldots \]

where \text{field}_\text{name} is the name of a field from the RECORDXX_SCHEMA definition that corresponds to this RECORDXX_SEQUENCE definition.

For example:

```sql
RECORD01_SCHEMA *
    \( \text{field}_1 \text{ smallint not null default } -1, \)
    \( \text{field}_2 \text{ double default } 0.0, \)
    \( \text{field}_3 \text{ integer not null default } 0, \)
    \( \text{field}_4 \text{ float default } -1, \)
RECORD01SEQUENCE *(\text{field}_3, \text{field}_1)*
```

In this example, the unique identification number has two components, \text{field}_3 and \text{field}_1. When comparing two ticks to determine their order in the sequence, DSE first compares the values of \text{field}_3 for both ticks. If they are different, the tick with the lesser of the two values comes first in the sequence. If the values for \text{field}_3 are the same, DSE compares the values of \text{field}_1.

You can include as many fields from the RECORDXX_SCHEMA definition in a RECORDXX_SEQUENCE definition as necessary, with the following restrictions and requirements:

- Fields can only occur once. For example, "(\text{field}_1, \text{field}_1)" is not allowed.
- Each field must have both a NOT NULL clause and a WITH DEFAULT clause.
- The default value in each field’s WITH DEFAULT clause should be numerically less than any expected value for the field. For example, if values for \text{field}_1 start with 0 and increase numerically, set the default to -1. If the field is one of the text types (char or varchar), choose a default string whose collation order is lower than any expected string value.
- You must define a RECORDXX_SEQUENCE for each RECORDXX_SCHEMA definition if high availability is enabled (HA_ENABLE = 1).

---

**Shared memory configuration parameters**

Use the shared memory configuration parameters to define the shared memory area that DSE can use and to define how DSE purges data.

**ADD_SHMSIZE**

This parameter specifies the size of shared memory (in bytes) that DSE adds when the shared memory that is being used is greater than the percentage that is specified by PURGE_HWM.
Shared memory can be a maximum of the number of bytes that are specified by MAX_SHMSIZE.

ADD_SHMSIZE is a positive integer with a default value of 0. If ADD_SHMSIZE is 0, DSE does not add shared memory to the size that is specified by the SHMSIZE parameter.

**FLBS**

This parameter specifies the free-list buffer size for the free list of the tick pool. Each feed pre-allocates the number of ticks that are specified by the free list.

FLBS is a positive integer with a default value of 20,000.

*Recommendation:* Specify the average number of ticks that the feed receives per second.

**HASHSIZE**

This parameter specifies the number of hash buckets that are reserved for the symbol hash table. The size of HASHSIZE depends on the number of unique symbols that you expect.

HASHSIZE must be a positive integer greater than 0 and has a default value of 50,000.

*Recommendation:* The HASHSIZE should be greater than the number of unique symbols divided by 100.

**MAX_SHMSIZE**

This parameter specifies the maximum size of shared memory (in bytes) for DSE.

When DSE uses more than the percentage of memory that is specified by PURGE_HWM, it dynamically adds shared memory segments of size ADD_SHMSIZE until one of the following conditions is met:

- Less memory is being used than the percentage that is specified by PURGE_HWM.
- The amount of shared memory that is allocated is the value for MAX_SHMSIZE.

MAX_SHMSIZE should equal SHMSIZE + (n * ADD_SHMSIZE), where n is an integer. The maximum amount of shared memory that you can allocate depends on the value that is specified by the operating system:

- In a Linux environment this is defined by the shmmmax and shmall kernel parameters.
- In a Solaris environment this is defined by the shinfo_shmmax kernel parameter.
- In an AIX environment all memory can be used as shared memory.

Refer to your operating system documentation for information on how to configure these kernel parameters.

MAX_SHMSIZE is a positive integer with a default value of 120,000,000.
NUM_INTERVALS
This parameter specifies the number of intervals that comprise the rolling window of time, which is the minimum number of seconds during which data is kept in shared memory.

The REFRESH_INTERVAL parameter defines the length, in seconds, of these intervals:
Rolling window of time = NUM_INTERVALS * REFRESH_INTERVAL

or
Rolling window of time = Number of intervals * Number of seconds per interval

For example, if the NUM_INTERVALS parameter is set to 6, and the REFRESH_INTERVAL parameter is set to 10 (seconds), then the rolling window of time is 60 seconds. Every 10 seconds, DSE purges the data that is in the oldest 10 seconds in the window.

Note: Memory overhead is associated with each interval. Increasing NUM_INTERVALS decreases the amount of memory available for storing ticks.

NUM_INTERVALS is a positive integer with a default value of 1.

PURGE_HWM
This parameter specifies the percentage of memory usage at which DSE begins adding memory segments or purging data. This percentage represents the purge "high water mark." DSE uses the value of PURGE_HWM to determine when to add a shared memory segment.

If the percentage of memory that contains ticks is greater than the percentage that is specified by PURGE_HWM, DSE attempts to allocate a new shared memory segment. If it is unable to allocate a segment, DSE purges ticks that were saved to disk from shared memory until the percentage of memory that is being used is less than the percentage that is specified by PURGE_LWM. DSE purges the oldest memory bucket first, and continues to purge the oldest memory buckets until enough memory is freed.

The value of PURGE_HWM is a percentage that is greater than 0 and less than 100. The default value of PURGE_HWM is 90.

PURGE_LWM
This parameter specifies the percentage of memory usage at which DSE stops adding memory segments or purging data. This percentage represents the purge "low water mark."

If DSE is unable to allocate a shared memory segment, DSE purges ticks that were saved to disk from shared memory until the percentage of memory that is being used is less than the percentage that is specified by PURGE_LWM. DSE purges the oldest memory bucket first, and continues to purge the oldest memory buckets until enough memory is freed.

The value of PURGE_LWM is a percentage that is an integer greater than 0 and less than the value of PURGE_HWM. The default value of PURGE_LWM is 25.
**REFRESH_INTERVAL**

This parameter specifies the number of seconds of each interval that comprises the rolling window of time, which is the minimum number of seconds during which data is kept in shared memory.

The REFRESH_INTERVAL value and the NUM_INTERVALS value determine the length, in seconds, of the rolling window of time:

Rolling window of time = NUM_INTERVALS * REFRESH_INTERVAL

or

Rolling window of time = Number of intervals * Number of seconds per interval

For example, if the NUM_INTERVALS parameter is set to 6, and the REFRESH_INTERVAL parameter is set to 10 (seconds), then the rolling window of time is 60 seconds. Every 10 seconds, DSE purges the data that is in the oldest 10 seconds in the window.

**Exception:** DSE purges data from shared memory differently when both of the following conditions are true:

- DSE stores data at a rate that causes the data to exceed the amount of available memory (as determined by the rolling window of time).
- The maximum value of the MAX_SHMSIZE parameter is reached.

In this situation, data that is already saved to disk is removed from memory until the amount of used memory is no greater than the percentage that is specified by PURGE_LWM. DSE purges the oldest memory bucket first, and continues to purge the oldest memory buckets until enough memory is freed.

REFRESH_INTERVAL is a positive integer with a default value of 60.

**SECHDR_SIZE**

This parameter specifies the maximum number of symbols for which DSE can collect data.

**Attention:** Do not set this value too low because you will not be able to capture data for all of the necessary symbols. You will receive an error message if you run out of space for symbols. Increasing this value reduces the amount of space available in memory for ticks.

SECHDR_SIZE is a positive integer with an absolute maximum value of 2 147 483 647. The actual maximum value depends on the configuration. It has a default value of 90 000.

**SHMKEY**

This parameter specifies the shared memory key that DSE uses when it allocates shared memory segments. If you are using multiple instances of DSE on the same server, each instance must use a unique shared memory key.

The first shared memory segment that is allocated will use the shared memory key. Each time an additional shared memory segment must be allocated, SHMKEY is incremented by one. SHMKEY is a positive integer with a default value of 268 658 176.
**SHMSIZE**

This parameter specifies the initial size of shared memory for DSE in bytes.

The value for SHMSIZE can grow to the amount that is defined by MAX_SHMSIZE in increments of the value for ADD_SHMSIZE. When deciding on a value for SHMSIZE, consider aspects of the expected data (for example, the rate of incoming data, the size of each tick, and how long you want to keep the data in memory).

SHMSIZE is a positive integer with a default value of 120 000 000.

---

**Storage system configuration parameters**

DSE stores data in stores that are contained in a storage system. The storage system can be a relational database management system like a DB2 system, or it can be another kind of storage system. Use the storage system configuration parameters to configure the storage system.

The XX in these parameter names is the storage system number, and can be 00 to 99.

**Related concept:**

See Chapter 19, “Enabling error recovery for storage systems,” on page 131 for information about what happens when all attempts by DSE to retry failed storage system operations or reconnect to storage systems do not succeed.

**STORESYSXX_CONN**

This parameter specifies information about connecting to the storage system.

The STORESYSXX_CONN parameter value is a string that is interpreted by the storage system that you use.

If you use the DB2 storage system that is supplied with DSE, specify the name of the DB2 database with this parameter.

For example:

`STORESYS01_CONN "my_database"`

This parameter has a default value of null.

**STORESYSXX_HA**

This parameter specifies whether the storage system is configured for high availability support.

Valid parameter values are:

0 Support for high availability is not enabled
1 Support for high availability is enabled.

This parameter has a default value of 0.
STORESYSXX_INFO

This parameter specifies additional information that is interpreted by the storage system.

This parameter can be accessed by calling the rtl_storesys_get_info_* API functions that are described in Chapter 27, “DSE API reference,” on page 285.

This parameter is a string with a default value of null.

STORESYSXX_LIB_PATH

This parameter specifies the path to the shared library of the storage system that you use.

The value of STORESYSXX_LIB_PATH can be one of the following:

- An absolute path
- A file that is located in the DSE_INSTALL_DIR/lib directory
- A file that is located in a directory in the LD_LIBRARY_PATH (or LIBPATH for AIX) environment variable

Recommendation: Specify an absolute path if DSE is part of a configuration in which DSE is started automatically. For example, DSE can be started automatically on system startup, by a cron job, or in a high availability configuration.

The library must contain implementations for the generic-storage interface that is required by DSE. DSE loads the library when it starts and uses the library for all store operations for that storage system. If one of the generic-storage interface functions is not required by your storage system, you can omit that function and DSE does not call that function.

This parameter is required.

STORESYSXX_MSG_PATH

This parameter specifies the path of the message file for storage-system-specific messages.

The value of STORESYSXX_MSG_PATH can be one of the following:

- An absolute path
- A path that is relative to the directory that contains the DSE configuration file
- A file that is located in the DSE_INSTALL_DIR/msg directory

This parameter is required.

STORESYSXX_NAME

This parameter specifies the name of the storage system. The significance of the value of STORESYSXX_NAME depends on the storage system that you use.

You can name your test storage system any text string of your choice. However, to make your configuration file more readable, use a meaningful name like “db2” or “test”, depending on the storage system being used.

STORESYSXX_NAME is a required parameter.
**STORESYSXX_PASSWORD**

This parameter specifies the password. STORESYSXX_PASSWORD is an encrypted string for a password that is interpreted by the storage system that you use.

You must create the value for STORESYSXX_PASSWORD with the rtl_make_pass utility.

**Example:** The DB2 database system uses STORESYSXX_PASSWORD as the user password when it connects to the database.

This parameter has a default value of null.

**STORESYSXX_RECONNECT_COUNT**

This parameter specifies the maximum number of times that DSE can attempt to reconnect to a data store after a storage system connection failure.

During a storage system connection failure, the connection to the data store is lost or certain resources related to the storage system are broken.

This parameter has a default value of 0.

**STORESYSXX_RECONNECT_PERIOD**

This parameter specifies the number of seconds that DSE waits between attempts to reconnect to a data store.

This parameter has a default value of 0.

**STORESYSXX_RETRY_COUNT**

This parameter specifies the maximum number of times that DSE can retry a failed storage system operation.

This parameter has a default value of 0.

**STORESYSXX_RETRY_PERIOD**

This parameter specifies the number of seconds that DSE waits between attempts to retry a failed storage system operation.

This parameter has a default value of 0.

**STORESYSXX_USER**

This parameter specifies the user name. STORESYSXX_USER is a string for a user name that is interpreted by the storage system that you use.

This parameter has a default value of null.

**Example:** The DB2 database system uses STORESYSXX_USER as the user name when it connects to the database.
Store configuration parameters

DSE stores ticks and symbols from data feeds. For each store, you can set the store configuration parameters to customize data storage.

The XX in these parameter names is the store number, and can be 00 to 99.

**STOREXX_AUTHLEVEL**
This parameter specifies the authorization level that is required to access the store.

To access a store, the group authorization level must be greater than or equal to the store authorization level.

STOREXX_AUTHLEVEL is an integer with a maximum value of 2 147 483 647. It has a default value of 1.

**STOREXX_DISCARD_ENABLE**
This parameter enables the ability to discard bad data from DSE shared memory into a discard table.

Valid parameter values are:

- 0 The ability to discard bad data is disabled.
- 1 The ability to discard bad data is enabled.

STOREXX_DISCARD_ENABLE has a default value of 0.

See "Creating a discard table for bad data" on page 133 for more information.

**STOREXX_FLUSH_BATCH**
This parameter specifies the number of ticks that DSE flushes from memory to the store at the same time.

For example, if the storage system is the DB2 database system, the batch size refers to the number of ticks that are written to the database in a single transaction. DSE marks each batch with the rtl_store_flush_batch_begin and rtl_store_flush_batch_end functions of the storage system tick-flushing interface.

STOREXX_FLUSH_BATCH is an integer with a default value of 10 000.

**STOREXX_FLUSH_PERIOD**
This parameter specifies how often DSE flushes new ticks from memory to the store.

If a flush takes longer than the value of STOREXX_FLUSH_PERIOD, the next flush begins immediately.

STOREXX_FLUSH_PERIOD is an integer that is measured in seconds and has a default value of 10.

**STOREXX_INFO**
This parameter specifies additional information about the store that is interpreted by the storage system.
This information can be accessed by calling the `rtl_store_get_info_*` API functions that are described in Chapter 27, “DSE API reference,” on page 285.

**STOREXX_INSERT_BATCH_SIZE**

This parameter is deprecated. Use the `STOREXX_SYMBOL_BATCH` parameter instead.

**STOREXX_META_BATCH**

This parameter specifies the maximum number of symbols whose metadata changes are written to the data store at the same time.

For example, if the storage system is the DB2 database system, the batch size refers to the number of rows of metadata that are written to the database in a single transaction.

Note: This parameter is ignored for stores that use a simple schema.

`STOREXX_META_BATCH` is a positive integer with a default value of 1000.

**STOREXX_META_MODE**

This parameter determines how DSE reads symbol metadata from the data store and writes symbol metadata to the data store.

Valid parameter values are:

0  Disables the reading of symbol metadata from the store into shared memory when DSE starts. Disables the manual or automatic writing of symbol metadata from shared memory to the store.

1  Enables the reading of symbol metadata from the store into shared memory when DSE starts. Metadata that is in DSE shared memory can be written to the data store manually with the `rtlmode -w` command.

Note: When DSE receives new symbols, it persists the symbol metadata automatically to the store.

2  Enables the reading of symbol metadata from the store into shared memory when DSE starts. Metadata that is in DSE shared memory can be written automatically, or manually (with the `rtlmode -w` command).

Note: When DSE receives new symbols, it persists the symbol metadata automatically to the store.

Note: This parameter is ignored for stores that use a simple schema.

`STOREXX_META_MODE` has a default value of 0.

Related reference

“`RECORDXX_META_SCHEMA” on page 61

This parameter specifies the structure of the symbol metadata in DSE memory.

**STOREXX_META_PERIOD**

This parameter specifies the interval, in seconds, between automatic writes of symbol metadata from shared memory to the data store.
If the amount of time that DSE spends writing the symbol metadata is longer than the interval defined by STOREXX_META_PERIOD, the next automatic write of symbol metadata begins immediately.

**Note:** This parameter is ignored for stores that use a simple schema.

STOREXX_META_PERIOD is a non-negative integer. If STOREXX_META_PERIOD is 0, DSE continuously writes metadata from shared memory to the data store. STOREXX_META_PERIOD has a default value of 3600 (one hour).

**STOREXX_NAME**
This parameter specifies the name of the store.

STOREXX_NAME is a string. This parameter is required.

**STOREXX_NEWSYMBOLS**
This parameter specifies whether the store accepts new symbols.

Valid parameter values are:
1  The store accepts new symbols.
0  The store does not accept new symbols

**Note:** This parameter is ignored for stores that use a simple schema.

This parameter has a default value of 1.

**STOREXX_NODISKWRITE**
This parameter specifies whether to write ticks to disk during a flush.

Valid parameter values are:
0  The store writes ticks to disk
1  The store does not write to disk

You can set a store to write at run time by using `rtlmode -d`. When a store does not write to disk, DSE does not call the tick-flushing interface functions that are provided by the storage system. If you change STOREXX_NODISKWRITE during a flush, the change does not take effect until the next flush.

This parameter has a default value of 0.

**STOREXX_NUM_FLUSHERS**
This parameter specifies how many flushers DSE allocates to flush ticks from shared memory to the store.

Each flusher is a separate thread that flushes data.

STOREXX_NUM_FLUSHERS is an integer with a default value of 1.

**STOREXX_RECTYPE**
This parameter specifies the record type that is supported by the store.
STOREXX_RECTYPE must be a number between 0 and 99, and this value must correspond to the record number (the XX value) of the record configuration parameters that are described in “Record configuration parameters” on page 61. DSE uses STOREXX_RECTYPE to determine the record type and dependent fields, which allows the store to handle ticks without using type-specific user code.

This parameter is required.

**STOREXX_SCHEMA**

This parameter specifies the schema name for the symbols table and the ticks table.

If you want to use special characters, spaces, or mixed case letters in a schema name, you must follow the rules for delimited identifiers.

This parameter is required and does not have a default value.

*Related reference*

[Chapter 11, “Rules for delimited identifiers,” on page 85](#)

The DSE configuration file can contain delimited identifiers for some of the parameters. Follow these rules and guidelines when using delimited identifiers in the configuration file.

**STOREXX_SCHEMA_TYPE**

This parameter specifies the schema type of the store.

Valid parameter values are:

- **0** The store uses a master-detail relationship, containing entities and events with timestamps. The entities have associated metadata that can be used to keep state or other entity-related data.
- **1** The store uses a simple schema with event records that do not require a timestamp. This type of store does not have entities or associated metadata.

This parameter is optional and has a default value of 0.

**STOREXX_SYMBOL_BATCH**

This parameter specifies the maximum number of new symbols that are inserted into the store at the same time.

For example, if the storage system is the DB2 database system, the batch size refers to the number of rows of new symbols that are written to the database in a single transaction.

*Note:* This parameter is ignored for stores that use a simple schema.

The value of this parameter is a positive integer. The default value is 100.

**STOREXX_SYMBOL_SET_SIZE**

This parameter specifies the size (number of rows) of the rowset for symbols data or symbols metadata that is passed to the storage system.

*Note:* This parameter is ignored for stores that use a simple schema.
The value of the STOREXX_SYMBOL_SET_SIZE parameter is a positive integer and must be less than the value of the STOREXX_META_BATCH parameter. The default value is 100.

**STOREXX_SYMBOL_TABLE**

This parameter specifies the name of the symbols table.

The symbols table name has the format `schema.symbol_table`, where:

- `schema` is the value of the STOREXX_SCHEMA parameter.
- `symbol_table` is the value of the STOREXX_SYMBOL_TABLE parameter.

For example, if the storage system is the DB2 database system, and these parameters are defined as follows:

```
STORE01_SCHEMA='dse'
STORE01_SYMBOL_TABLE='trade_symbols'
```

Then the name of the symbols table is `dse.trade_symbols`.

If you want to use special characters, spaces, or mixed case letters in a table name, you must follow the rules for delimited identifiers.

**Note:** This parameter is ignored for stores that use a simple schema.

This parameter is required and does not have a default value.

**Related reference**

[Chapter 11, “Rules for delimited identifiers,” on page 85](#)

The DSE configuration file can contain delimited identifiers for some of the parameters. Follow these rules and guidelines when using delimited identifiers in the configuration file.

**STOREXX_SYSNAME**

This parameter specifies the name of the parent storage system.

STOREXX_SYSNAME is a string that corresponds to the STORESYSXX_NAME entry of the storage system.

This parameter is required.

**STOREXX_TICK_SET_SIZE**

This parameter specifies the size (number of rows) of the rowset for ticks data that is passed to the storage system.

The value of the STOREXX_TICK_SET_SIZE parameter is a positive integer and must be less than the value of the STOREXX_FLUSH_BATCH parameter. The default value is 1000.

**STOREXX_TICK_TABLE**

This parameter specifies the name of the ticks table.

The ticks table name has the format `schema.tick_table`, where:

- `schema` is the value of the STOREXX_SCHEMA parameter.
- `tick_table` is the value of the STOREXX_TICK_TABLE parameter.
For example, if the storage system is the DB2 database system, and these parameters are defined as follows:

```
STORE01_SCHEMA='dse'
STORE01_TICK_TABLE='trade_ticks'
```

Then the name of the ticks table is `dse.trade_ticks`.

If you want to use special characters, spaces, or mixed case letters in a table name, you must follow the rules for delimited identifiers.

This parameter is required and does not have a default value.

**Related reference**

- [Chapter 11, “Rules for delimited identifiers,” on page 85](#)

The DSE configuration file can contain delimited identifiers for some of the parameters. Follow these rules and guidelines when using delimited identifiers in the configuration file.

### Trace configuration parameters

Use the trace configuration parameters to set up tracing for the components of DSE.

**Related concepts**

- [Chapter 31, “DSE traces,” on page 485](#)

DSE provides a trace mechanism to aid in troubleshooting problems. Set up the trace file and settings using the trace configuration parameters.

#### TRACEBUFFER_SIZE

This parameter specifies the size (in bytes) of the buffer in shared memory that stores trace information.

**Restriction:** The value of the TRACEBUFFER_SIZE parameter cannot exceed 100 MB, or 10% of the value of the SHMSIZE parameter, whichever is less.

The minimum value of TRACEBUFFER_SIZE is 100 KB. If the value of this parameter is less than 100 KB, it is changed to 100 KB.

**Example:**

```
TRACEBUFFER_SIZE 256000
```

If the TRACEBUFFER_SIZE parameter is not defined, DSE uses the default size of 512 KB of shared memory for storing the trace information.

#### TRACEFILE_DIR

This parameter specifies the directory that contains the trace file.

**Example:**

```
TRACEFILE_DIR "/tmp/myuser"
```

If the TRACEFILE_DIR parameter is not defined, the default directory is the running directory of DSE.
TRACEFILE_SIZE

This parameter specifies the size (in bytes) of the trace file.

Example:

```
TRACEFILE_SIZE 50000000
```

The TRACEFILE_SIZE parameter has a default value of 5 MB.

TRACELEVEL_component

Use the trace level configuration parameters to enable and configure tracing for specific components of DSE.

See [“Trace level configuration parameters”](#) for information on these parameters.

---

Trace level configuration parameters

Use the trace level configuration parameters to enable tracing for the components of DSE and specify the trace level for each component. By default, tracing is turned off.

Each of the trace level configuration parameters in Table 10 on page 81 can have one of the following trace level settings. These settings are not case-sensitive.

**High**

High-level tracing provides a high-level view of the execution of specified components. Of the trace levels, high-level tracing produces the least amount of trace information. High-level tracing information includes entry and exit points of the major functions within the specified components of DSE.

**Med**

Medium-level tracing provides a more detailed view of the functions that are run within the specified components. Medium-level tracing produces more information than high-level tracing and includes the information from high-level tracing. Medium-level tracing information includes significant information and states within major functions, and entry and exit points of lower-level functions within the specified components of DSE.

**Low**

Low-level tracing provides the most detailed review of the functions that are run within the specified components. Of the trace levels, low-level tracing produces the most amount of information. Low-level tracing information includes all trace information within the specified components of DSE. The volume of trace information that is written at this level might impact performance.

**Off**

Tracing is off. (Default)

You can enable tracing for individual components of DSE, or you can enable tracing for all of the components. You can specify different trace level settings for each of the components of DSE. If multiple trace level settings are defined in the configuration file, the last parameter definition takes precedence over the previous parameter definitions.

In the following example, high-level tracing is enabled for all of the components of DSE except the feed handler. Low-level tracing is enabled for the feed handler.

```
TRACELEVEL_ALL HIGH
TRACELEVEL_HANDLER LOW
```
In the following example, the last parameter definition overrides the previous parameter definitions. Trace level is set to medium for all components.

TRACELEVEL_MAIN HIGH  
TRACELEVEL_FEED LOW   
TRACELEVEL_ALL MED

You can dynamically enable, disable, and change the trace levels for any component while DSE is running by running the rtlmode -T command.

Table 10. Trace level configuration parameters for components

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACELEVEL_ALL</td>
<td>Specifies the trace level for all of the DSE threads and components. This trace level applies to any component whose trace level is not explicitly set by a trace level configuration parameter.</td>
</tr>
<tr>
<td>TRACELEVEL_FEED</td>
<td>Specifies the trace level for the feed thread that processes the feed and invokes the feed handler library.</td>
</tr>
<tr>
<td>TRACELEVEL_FLUSHER</td>
<td>Specifies the trace level for the fluser thread that writes ticks from memory to the persistent store.</td>
</tr>
<tr>
<td>TRACELEVEL_HA</td>
<td>Specifies the trace level for the high availability component of DSE.</td>
</tr>
<tr>
<td>TRACELEVEL_HANDLER</td>
<td>Specifies the trace level for the user-defined feed handler library.</td>
</tr>
<tr>
<td>TRACELEVEL_HA_COMM_LIB</td>
<td>Specifies the trace level for a user-defined high availability communication library.</td>
</tr>
<tr>
<td>TRACELEVEL_INSERTER</td>
<td>Specifies the trace level for the inserter thread that manages new symbols in the store.</td>
</tr>
<tr>
<td>TRACELEVEL_LISTENER</td>
<td>Specifies the trace level for the thread that processes rtlmode commands.</td>
</tr>
<tr>
<td>TRACELEVEL_MAIN</td>
<td>Specifies the trace level for the main program, including startup and shutdown.</td>
</tr>
<tr>
<td>TRACELEVEL_METAWRITER</td>
<td>Specifies the trace level for the metawriter thread that writes symbol metadata from memory to the persistent store.</td>
</tr>
<tr>
<td>TRACELEVEL_PARTITIONER</td>
<td>Specifies the trace level for the partitioner thread that allocates ticks to fluser threads.</td>
</tr>
<tr>
<td>TRACELEVEL_PURGER</td>
<td>Specifies the trace level for the purger thread that removes old ticks from memory, to create space for new ticks.</td>
</tr>
<tr>
<td>TRACELEVEL_QUERY</td>
<td>Specifies the trace level for the query thread that processes API requests.</td>
</tr>
<tr>
<td>TRACELEVEL_STORESYS</td>
<td>Specifies the trace level for the storage system library.</td>
</tr>
<tr>
<td>TRACELEVEL_TRANSPORT</td>
<td>Specifies the trace level for the transport.</td>
</tr>
</tbody>
</table>

Related concepts

Chapter 31, “DSE traces,” on page 485

DSE provides a trace mechanism to aid in troubleshooting problems. Set up the trace file and settings using the trace configuration parameters.
Transport configuration parameters

A transport reads data from a feed source and writes data to a feed target. Feed sources and targets can be files, sockets, or queues. Use these parameters to configure each transport.

The XX in these parameter names is the transport number, and can be 00 to 99.

**TRANSPORTXX_INFO**

This optional parameter specifies additional information to be interpreted by the transport.

This information can be accessed by calling the `rtl_transport_get_info` function.

Related concepts

“TRANSPORTXX_INFO parameter for pre-built transports” on page 172

You can modify the behavior of each pre-built transport by specifying additional information in the TRANSPORTXX_INFO parameter.

**TRANSPORTXX_LIB_PATH**

This parameter specifies the path of the shared library for a transport.

The value of the TRANSPORTXX_LIB_PATH parameter can be one of the following:

- An absolute path
- A file that is located in the `DSE_INSTALL_DIR/lib` directory
- A file that is located in a directory in the `LD_LIBRARY_PATH` (or `LIBPATH` for AIX) environment variable

This parameter is required.

**TRANSPORTXX_MSG_PATH**

This parameter specifies the path of the message catalog for the transport.

The value of the TRANSPORTXX_MSG_PATH parameter can be one of the following:

- An absolute path
- A path that is relative to the directory that contains the DSE configuration file
- A file that is located in the `DSE_INSTALL_DIR/msg` directory

This parameter is required.

**TRANSPORTXX_NAME**

This parameter specifies a text string that uniquely identifies a transport.

The name is used to associate a transport with a feed that is identified with the FEEDXX_TRANSPORT_NAME parameter.

This parameter is required.
Chapter 10. Supported data types

DSE provides or interacts with different data type systems that developers can use when working with different kinds of data streams and storage systems.

The type systems are:

**DSE record schema type**
Used in the configuration file for tick and metadata definitions.

The RECORDXX_SCHEMA specifies the structure of ticks and the RECORDXX_META_SCHEMA specifies the structure of the symbol metadata in DSE memory.

**DSE field type (rtl_field_type)**
Used in the field information data structures for ticks and metadata records that are used by C and Java developers to determine field types at runtime.

**C programming language type**
Used for programming in C.

**Java programming language type**
Used for programming in Java.

**DB2 SQL type**
Used for the underlying DB2 database storage system table definitions and wrapper nickname definitions.

The following table lists the data types that DSE supports and shows the relationship with equivalent types in the other type systems.

<table>
<thead>
<tr>
<th>DSE record schema type</th>
<th>DSE field type (rtl_field_type)</th>
<th>C type</th>
<th>Java type</th>
<th>DB2 SQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallint</td>
<td>RTL_TYPE_INT16</td>
<td>short</td>
<td>short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>bigint</td>
<td>RTL_TYPE_INT64</td>
<td>long</td>
<td>long</td>
<td>BIGINT</td>
</tr>
<tr>
<td>int</td>
<td>RTL_TYPE_INT32</td>
<td>int</td>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>double</td>
<td>RTL_TYPE_DOUBLE</td>
<td>double</td>
<td>double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>double precision</td>
<td></td>
<td></td>
<td></td>
<td>FLOAT(n),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>where</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float(n), where</td>
<td></td>
<td></td>
<td></td>
<td>25 &lt;= n &lt;= 53</td>
</tr>
<tr>
<td>25 &lt;= n &lt;= 53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 11. Supported data types (continued)

<table>
<thead>
<tr>
<th>DSE record schema type</th>
<th>DSE field type (rtl_field_type)</th>
<th>C type</th>
<th>Java type</th>
<th>DB2 SQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallfloat</td>
<td>RTL_TYPE_FLOAT</td>
<td>float</td>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>real</td>
<td></td>
<td></td>
<td></td>
<td>FLOAT(n), where</td>
</tr>
<tr>
<td>float(n), where</td>
<td></td>
<td></td>
<td></td>
<td>1 &lt;= n &lt;= 24</td>
</tr>
<tr>
<td>timestamp(^i)</td>
<td>RTL_TYPE_RTL_DATETIME</td>
<td>rtl_datetime</td>
<td>com.ibm.rtl.api.RTLDateTime</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>timestamp(rtl_datetime)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time(^t)</td>
<td>RTL_TYPE_TIME_T</td>
<td>time_t</td>
<td>java.util.Date</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>timestamp(time_t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>timestamp(rtl_java_date)</td>
<td>RTL_TYPE_RTL_JAVA_DATE</td>
<td>rtl_java_date</td>
<td>com.ibm.rtl.api.RTLDateTime</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>date</td>
<td>RTL_TYPE_RTL_DATE</td>
<td>rtl_dateTime</td>
<td>com.ibm.rtl.api.RTLDateTime</td>
<td>DATE</td>
</tr>
<tr>
<td>time</td>
<td>RTL_TYPE_RTL_TIME</td>
<td>rtl_dateTime</td>
<td>com.ibm.rtl.api.RTLDateTime</td>
<td>TIME</td>
</tr>
<tr>
<td>char(n)</td>
<td>RTL_TYPE_CHAR</td>
<td>char[n+1]</td>
<td>java.lang.String</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>character(n)</td>
<td></td>
<td></td>
<td></td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>character varying(n)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>varchar(n)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte(n)</td>
<td>RTL_TYPE_BYTE</td>
<td>rtl_byte</td>
<td>byte[n]</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOR BIT DATA</td>
</tr>
<tr>
<td>boolean</td>
<td>RTL_TYPE_BOOLEAN</td>
<td>rtl_boolean</td>
<td>boolean</td>
<td>CHAR(1) FOR BIT DATA</td>
</tr>
</tbody>
</table>

Note:
1. DSE record schema types timestamp and time\(_t\) are deprecated. You should use timestamp(rtl_datetime) and timestamp(time\(_t\)) instead.

Related reference

“RECORDXX_SCHEMA” on page 65
This parameter specifies the structure of ticks in DSE memory.

“RECORDXX_META_SCHEMA” on page 61
This parameter specifies the structure of the symbol metadata in DSE memory.

“rtl_get_tick_info()” on page 334
This function gets information about the tick for a record type.

“rtl_get_meta_info()” on page 326
This function gets information about the metadata for a record type.
Chapter 11. Rules for delimited identifiers

The DSE configuration file can contain delimited identifiers for some of the parameters. Follow these rules and guidelines when using delimited identifiers in the configuration file.

A delimited identifier:
- Is enclosed within single or double quotation marks
- Can include uppercase and lowercase letters, numbers, specific characters, and spaces
- Can contain a double quotation mark, represented by two consecutive double quotation marks: ""

Delimited identifiers can be:
- Store names, as defined by the STOREXX_SCHEMA, STOREXX_SYMBOL_TABLE, or STOREXX_TICKS_TABLE parameters
- Fields in the record schema definition, as defined by the RECORDXX_META_SCHEMA or RECORDXX_SCHEMA parameters

The following rules and guidelines apply to delimited identifiers:
- The quotation marks in a delimited identifier must be preceded by the backslash escape character (\).
- Identifiers that are enclosed in quotation marks are case sensitive.
- The following characters can be part of an identifier and can optionally be escaped by a backslash (\):
  . Period
  # Number sign or hash sign
  : Colon
  ; Semicolon
  ! Exclamation point
  @ At sign
  $ Dollar sign
  % Percent sign
  ^ Caret
  & Ampersand
  * Asterisk
  | Pipe
  - Hyphen
  + Plus sign
  = Equal sign
  [ ] Brackets
  { } Braces
/  Forward slash
<  Less than symbol
>  Greater than symbol
~  Tilde
`  Grave accent
?  Question mark

- The following characters can be part of an identifier and must be escaped by a backslash (\):
  '  Single quotation mark
  "  Quotation mark
  (  Opening parenthesis
  )  Closing parenthesis
  \  Backslash
  Space
  Tab

- Any character that is escaped by a preceding backslash is considered to be part of the identifier. DSE removes the backslash escape characters before passing the values to the storage system.

- Strings enclosed in single quotation marks are literal strings. Any characters in the string are considered to be part of the string. Two consecutive single quotation marks can be used in the string.

**Example 1:**

In this example, the store name is "Mixed.Case Store" and the corresponding definition in the configuration file is:
STORE01_NAME  "\"Mixed.Case\ Store\"

DSE passes the value "Mixed.Case Store" to the storage system.

**Example 2:**

In this example, the record schema definition is:
RECORD01_SCHEMA "(  
  field1 timestamp,
  field2 double with null -1,
  "Case\ Sensitive.Field\" int with null -1,
  ...)

DSE passes the value "Case Sensitive.Field" to the storage system.

**Related concepts**
Chapter 9, “System configuration parameters,” on page 51

The DSE configuration file contains configuration parameters that describe your system and the data that you are reading. DSE provides sample configuration files.
Chapter 12. Restartable mode for high availability

DSE can be run in restartable or non-restartable mode. In restartable mode, DSE guards against data loss in case of system failure.

During normal DSE operations (non-restartable mode), DSE caches data in shared memory before flushing it to the storage system. If an unplanned outage occurs, data in shared memory that is not yet flushed to a storage system can be lost and unrecoverable.

In restartable mode, DSE provides a transactional link between the feed source and the storage system. In other words, DSE manages feed reads and store writes as a single distributed, or global, transaction. If an unplanned outage occurs, changes are either committed (if the transaction is completed before the outage) or rolled back. Data is not lost or compromised.

DSE can be run in only one of the modes, and you cannot switch modes after you start DSE. When deciding between running DSE in restartable or non-restartable mode, consider the restrictions that are described in the following table.

Table 12. Comparing DSE in restartable and non-restartable mode

<table>
<thead>
<tr>
<th>DSE in restartable mode</th>
<th>DSE in non-restartable mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ticks data is recoverable if a system failure occurs. Symbols data and metadata is not recoverable.</td>
<td>Data might not be recoverable if a system failure occurs.</td>
</tr>
<tr>
<td>Only feeds using a transport that supports distributed (two-phase commit) transactional characteristics, such as WebSphere MQ, are supported.</td>
<td>All feed source types are supported.</td>
</tr>
<tr>
<td>Ticks in shared memory cannot be queried.</td>
<td>Ticks in shared memory can be queried.</td>
</tr>
<tr>
<td>Performance is diminished due to restrictions and overhead of distributed (two-phase commits) transactions.</td>
<td>Performance is unaffected.</td>
</tr>
<tr>
<td>Each flusher thread can persist tick data from at most 1 feed handler.</td>
<td>Each flusher thread can persist tick data from more than 1 feed handler.</td>
</tr>
</tbody>
</table>

DSE (in either restartable or non-restartable mode) can read data from transactional feed sources, such as WebSphere MQ queues, and can persist the data from that feed to the DB2 database system. While running DSE in restartable mode, the following runtime restrictions and requirements apply:

- During the time between system failure and restart of DSE, you cannot:
  - Repartition the database.
  - Make schema changes to the database.
  - Change any DSE configuration parameters that are related to the transactional feed source.
- WebSphere MQ must act as the transaction manager, managing the message queues. You cannot change queue names or other WebSphere MQ parameters during a restart.
To run DSE in restartable mode, set the RESTART_ENABLE parameter to 1 in the configuration file before starting DSE. See "General system configuration parameters" on page 52 for more information.

DSE provides a scenario to demonstrate how to run DSE in restartable mode with WebSphere MQ as the feed source and DB2 as the storage system. See Chapter 13, “Scenario: Running DSE in restartable mode with WebSphere MQ and DB2,” on page 89 for information about this scenario.

**Related reference**

"High availability configuration parameters" on page 57

Use these parameters to configure a high availability system that consists of multiple DSE instances.
Chapter 13. Scenario: Running DSE in restartable mode with WebSphere MQ and DB2

This scenario demonstrates how to run DSE in restartable mode, using WebSphere MQ for AIX, Linux, and Solaris, Version 6.0 as the transactional feed source and DB2 Version 9.5 for Linux, UNIX, and Windows as the storage system. Follow the steps in this scenario to understand how to set up a restartable data loading system.

In this scenario, you install and configure the software that is needed to run the scenario, run DSE to read messages from WebSphere MQ and persist the data to a DB2 database, simulate a system failure and recover from the failure, and then shut down the system. Figure 1 shows the configuration of DSE, WebSphere MQ, and the DB2 database system that is used in this scenario.

In this scenario, you create a queue manager (venus.queue.manager) and a queue (ORANGE.QUEUE) in WebSphere MQ. Then you populate the queue with test data, using the mqsend feed simulator program that DSE provides. The pre-built WebSphere MQ transport reads messages from the queue. The simple feed handler that is provided by DSE gets the messages from the transport and extracts the ticks data. DSE persists the data to the TESTDB DB2 database. As the data moves from the feed to the database, it is handled as a transaction with two-phase commit control to protect against data loss if a system failure occurs.

This scenario includes the following tasks:
- “Installing the software” on page 90
- “Setting up DSE” on page 90
- “Setting up the DB2 database system” on page 91
Installing the software

Install the software components (DSE, WebSphere MQ, and the DB2 database system) that this scenario uses.

To install the software:
1. Install DSE. Follow the instructions in Chapter 4, “Installing DSE,” on page 11.
2. Install the following WebSphere MQ components:
   - Runtime
   - SDK
   - Server
   - Sample programs
   - Man pages
   For detailed information, refer to the instructions for installing a WebSphere MQ server in the Quick Beginnings guide for your platform. This information is available in the WebSphere MQ information center at http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/
3. Install the DB2 database system.

Setting up DSE

To set up DSE for this scenario, edit the configuration file, then build a WebSphere MQ feed simulator and simple feed handler.

DSE provides the mqsend program to simulate the data feed. The simple feed handler is included in the product.

To set up DSE:
1. Ensure that the DSE and DB2 environment variables are set up. See “Setting up the environment variables” on page 18.
2. Create a copy of the sample rtlconfig.db2 configuration file in the DSE_INSTALL_DIR/examples/restart/config subdirectory, and give the new file a name (for example, rtlconfig.dse).
3. Edit your copy of the configuration file and replace the following parameter values with ones that are appropriate for your system:
   - **SHMKEY**
     Replace DSE1_SHMKEY with an integer that represents a unique shared memory key to be used (for example, 268658432).
   - **RTL_SERVICE_NAME**
     Replace DSE1_PORT with the port number to which a client application (such as the wrapper) connects (for example, 9324).
     **Restriction:** This number cannot be used by any other application.
   - **STORESYS01_CONN**
     Replace TESTDB if your test database has a different name.
4. Build the data feed simulator by running the following commands:
cd $DSE_INSTALL_DIR/examples/restart/driver
make

This creates the mqsend feed simulator program.

5. Build the simple feed handler by running the following commands:
cd $DSE_INSTALL_DIR/examples/simple/handler
make

This creates the simplehandler.so library file and saves it in the $DSE_INSTALL_DIR/lib directory.

---

**Setting up the DB2 database system**

To run this scenario, create a sample database to store the tick data from the simulated data feed.

The sample database should have a ticks table and a symbols table.

To set up the DB2 database system:
1. Start the DB2 database system.
2. Create a DB2 database named TESTDB.
3. Create the ticks table and symbols table by following the instructions in "Setting up a store" on page 20.

---

**Setting up and starting WebSphere MQ**

Configure WebSphere MQ for this scenario. Establish administrator authority and set up the WebSphere MQ environment.

To configure WebSphere MQ:
1. Add your user name to the mqm group, the group of WebSphere MQ administrators. Doing so gives you authority to start and stop the queue managers.
2. Create a new queue manager named venus.queue.manager by running the following WebSphere MQ command:
crtmqm venus.queue.manager
3. Edit the queue manager configuration file, qm.ini, to set up WebSphere MQ as the transaction manager in this scenario. This file contains information that is relevant to a specific queue manager. The qm.ini file is automatically created when the queue manager with which it is associated is created. To edit the qm.ini file:
   a. Locate the file in the following subdirectory:
      /var/mqm/qmgrs/venus!queue!manager/
   b. Add XAResourceManager stanzas to the qm.ini file. You can use the sample qm.ini file in the DSE_INSTALL_DIR/examples/restart/mqm subdirectory as a guide. For a single partitioned DB2 database, only one XAResourceManager stanza is required. For example:

```yaml
XAResourceManager:
  Name=DB2_testdb
  SwitchFile=db2swit
  XAOpenString=db=testdb,uid=db2inst2,pwd=dse4test,toc=t
  XACloseString=
  ThreadOfControl=THREAD
```
The XAOpenString definition provides the database name, user name and password that WebSphere MQ uses to establish a database connection. The toc=t and ThreadOfControl=THREAD definitions allow multiple concurrent transactions to be run in parallel.

For other database configurations, DSE provides sample qm.ini files with appropriate XAResourceManager stanzas in the DSE_INSTALL_DIR/examples/restart/mqm directory:

smp_qm.ini
For multi-partitioned DB2 databases on a single host system.

mpp_qm.ini
For multi-partitioned DB2 databases across multiple host systems.

4. Compile the DB2 XA switch load file. WebSphere MQ provides the db2swit.c file, which links WebSphere MQ to the DB2 database system for two-phase commit transactions that are managed by WebSphere MQ.
   a. Change to the following directory, which contains the xaswit.mak file:
      /opt/mqm/samp/xatm
   b. You might need to update the one or both DB2LIBPATH variable values in the xaswit.mak file to reflect the location of your DB2 libraries. For example, on a 64-bit Linux system, you might need to make these changes:
      DB2LIBPATH32=/opt/IBM/db2/v9.5/lib
      DB2LIBPATH64=/opt/IBM/db2/v9.5/lib64
   c. To make the DB2 XA switch load file, run the following command:
      make -f xaswit.mak db2swit
      A DB2 XA switch load file is created and copied to one of the following WebSphere MQ product directories (depending on whether the DB2 instance is 32-bit or 64-bit):
         • MQEXITDIR32=/var/mqm/exits
         • MQEXITDIR64=/var/mqm/exits64

5. Start the queue manager by running the following WebSphere MQ command
   strmqm venus.queue.manager

6. Create a queue called ORANGE.QUEUE by running the following WebSphere MQ command:
   runmqsc venus.queue.manager <<ZZZZZZ
   define qlocal(ORANGE.QUEUE) maxdepth(100000)
   end
   ZZZZZZ
   This command creates a queue with a maximum depth of 100,000 messages.

   Requirement: The queue manager name and queue name must match the FEEDXX_SOURCE parameter value in the DSE configuration file. Note that these names are case-sensitive. For example, the configuration file in the examples/restart/config subdirectory must contain the following definition:
   FEED01_SOURCE "venus.queue.manager:ORANGE.QUEUE"

WebSphere MQ creates a log file to store errors that are encountered. Check the log file in the following location if any errors occur:
/var/mqm/qmgrs/venus!queue!manager/errors/*.log
Starting DSE and the feed simulator

You can begin loading data. For this scenario, you load 100,000 ticks into the database slowly, over a period of 300 seconds, allowing you to monitor the load process and simulate a DSE failure and restart.

To start DSE and the feed simulator:
1. Start DSE using the rtlconfig.dse configuration file that you edited in "Setting up DSE" on page 90, with the following command:
   
   ```sh
   $DSE_INSTALL_DIR/bin/rtloader -c rtlconfig.dse &
   ```

   DSE waits indefinitely for messages to arrive on the source queue.

2. Start the mqsend feed simulator with the following command:
   
   ```sh
   $DSE_INSTALL_DIR/examples/restart/driver/mqsend \  
   -Ovenus.queue.manager:ORANGE.QUEUE \  
   -f$DSE_INSTALL_DIR/examples/restart/driver/tick_data \  
   -r100 -n100 -T0.3 &
   ```

   These options cause mqsend to read the tick_data file (which contains 1000 ticks) 100 times and place ticks on the queue in batches of 100, with a delay of 0.3 seconds between each batch.

   For additional information on mqsend options, run the mqsend -help command or examine the mqsend.c source code.

3. Monitor DSE and the DB2 TESTDB database to confirm that ticks are flowing through DSE and are being stored in the database.
   a. Run the rtlstat -d command to view counts of ticks that were processed by the feed thread.
   b. Run the rtlstat -P command to view partitioning counts.
   c. Run the rtlstat -f command to view counts of ticks that were flushed to the database.
   d. Execute the following SQL statement with the DB2 command line processor to view the number of ticks stored in the database table:

   ```sql
   SELECT COUNT(*) FROM rtl_ticks_ticks
   ```

Recovering from a simulated system failure

Simulate a DSE failure to confirm that you can recover from this system failure without any loss of data.

When DSE is running in restartable mode, reading data from a WebSphere MQ queue and writing it to a DB2 database is treated as a distributed transaction, with two-phase commit. If DSE is terminated while data is being transferred from a WebSphere MQ queue to a DB2 database, all active transactions are automatically rolled back to the last commit point upon restart of DSE. All messages whose ticks had not been persisted to the database are read again and no data is lost or duplicated.

To simulate a failure and recover from it:
1. Terminate DSE by running the following command:
   
   ```sh
   kill -9 rtloader_pid
   ```

   Where rtloader_pid is the process ID of the running DSE instance.

2. Examine how many ticks were loaded into the database before DSE was terminated by executing the following SQL statement with the DB2 command line processor:
SELECT COUNT(*) FROM rtl.ticks_ticks

3. Restart DSE with the following command:
   $DSE_INSTALL_DIR/bin/rtloader -f -c rtlconfig.dse &
   The -f option forces DSE to reinitialize its shared memory.

4. Monitor the flow of ticks through DSE into the database.
   a. Run the rtlstat -d command to view counts of ticks that were processed
      by the feed thread.
   b. Run the rtlstat -P command to view partitioning counts.
   c. Run the rtlstat -f command to view counts of ticks that were flushed to
      the database.
   d. Execute the following SQL statement with the DB2 command line processor
      to view the number of ticks stored in the database table:
         SELECT COUNT(*) FROM rtl.ticks_ticks

5. When all 100 000 ticks are placed on the queue by the mq_send program, the
   program exits. You can verify that all 100 000 ticks were loaded into the
   database by:
   • Executing the following SQL statement:
     SELECT COUNT(*) FROM rtl.ticks_ticks
   • Running the following WebSphere MQ command:
     runmqsc venus.queue.manager <ZZZ
     display qlocal(ORANGE.QUEUE)
     end
     ZZZ

   When the queue is empty, its depth is 0.

---

**Shutting down the system**

Shut down the software components of this scenario after you have finished
running it.

To shut down the software components:

1. Shut down DSE using the rtlmode utility program:
   $DSE_INSTALL_DIR/bin/rtlmode -s shmkey -ky
   Where *shmkey* is the value of the SHMKEY parameter in the DSE configuration
   file.

2. Shut down the WebSphere MQ queue manager by running the following
   WebSphere MQ command:
   endmqm -i venus.queue.manager

3. Stop the DB2 database system by running the following DB2 commands:
   db2 force application all
   db2stop
Chapter 14. High availability clusters

A high availability cluster, consisting of multiple host systems and peer DSE instances, ensures continuous availability of data if a failure occurs.

In a simple DSE configuration that has one DSE instance running on a host system, a failure of the DSE instance or of the system can result in loss of data. This loss of data occurs when data in memory is not persisted to permanent storage (for example, a database) before the failure occurs.

A high availability configuration consists of two or more host systems, each of which has a running DSE instance. This group of multiple host systems and peer DSE instances comprise a high availability cluster. Multiple peer DSE instances provide failover capability. One DSE instance is the live instance, and it performs the functions of DSE. The other instances are standby instances. They are synchronized with the live DSE instance and are ready to take over the functions of the live instance if a failure occurs.

If a DSE instance fails, it can be recovered and brought back into the high availability cluster as a standby instance. One of the standby instances takes over the functions of the live instance while the failed instance is recovered and restarted, ensuring continuous availability.

Restriction

High availability is supported for symbol and metadata information only. High availability for tick data is not supported in this release.

Requirements:

- You need cluster management software to detect failures and manage the failover process. See “High availability cluster management software” on page 97 for more information.
- Each system in a high availability cluster must be identical. Each DSE instance in a cluster must have identical configurations and receive identical data feeds. See “Configuring and starting a high availability cluster” on page 96 for more information.

DSE provides a scenario to demonstrate how to build and manage a high availability cluster. See “Scenario: Building and managing a high availability cluster,” on page 101 for information about this scenario.

Related reference

“High availability configuration parameters” on page 57
Use these parameters to configure a high availability system that consists of multiple DSE instances.

High availability states

A DSE instance in a high availability cluster is in one of various states that indicate its status. The states of DSE instances in a cluster can change dynamically to ensure high availability if the live instance fails.
In a high availability cluster with one live DSE instance and one standby DSE instance, the live instance and standby instance are synchronized to each other. The live instance is in ACTIVE state, and the standby instance is in STANDBY state. If the live instance fails, the cluster management software that you are using can put the standby instance into ACTIVE state, so that it becomes the live instance. It performs the functions of the live instance, and completes any in-progress work of the failed live instance.

Each of the DSE instances in a high availability cluster is in one of the following states:

**ACTIVE**
The DSE instance is the live instance. Only one instance can be the live instance.

**INACTIVE**
The DSE instance is unavailable, because it failed or is stopped.

**STANDBY**
The DSE instance is a standby instance. It is synchronized with the live instance and is ready to take over if the live instance fails.

**STARTED**
The DSE instance that is a standby instance that is not synchronized with the live instance. A standby instance is in STARTED state when either:

- The standby instance did not connect to the live instance to initiate synchronization.
- The connection between the standby instance and the live instance is broken, and therefore, the synchronization between the instances is broken.

When a standby instance connects to a live instance, it is in STARTED state, but after the connection is established, the standby instance is in SYNCHRONIZING state automatically. When the synchronization is complete, the standby instance is in STANDBY state.

**SYNCHRONIZING**
The standby DSE instance is in the process of synchronizing with the live instance.

---

**Configuring and starting a high availability cluster**

Configure a DSE high availability cluster to utilize the failover capability and prevent loss of data that result from system failures.

To configure and start a DSE high availability cluster, complete the following steps:

1. Set up a high availability cluster of two or more host systems. Ensure that the high availability cluster meets the following requirements:
   - Each host system must:
     - Have a running DSE instance.
     - Be physically separate from the other systems, with separate or redundant power supplies.
     - Have the same hardware architecture (preferably identical).
     - Run the same operating system (same version and preferably the same patch level).
     - Have its own network host name and IP address.
- Have redundant communication paths between host systems.
- Each DSE instance in a high availability cluster must have an identical configuration and receive identical data feeds. Use identical copies of the same configuration file to configure each DSE instance.
- Each message from the data feed must have a unique identifier, such as a sequence number.

2. For each instance, define the high availability configuration parameters in the DSE configuration file. Ensure that the HA_ENABLE configuration parameter is set to 1, to enable the high availability function. See “High availability configuration parameters” on page 57 for information about the high availability configuration parameters.

3. Install and set up software to manage your high availability cluster. See “High availability cluster management software” for more information.

4. Start the high availability cluster, using one of the following methods:
   - Start all of the DSE instances in STANDBY state. Then select one instance to be the live instance, and change the state of that instance to ACTIVE by issuing the rtlmode -M I command or by using the rtl_state_set function.
     After a live DSE instance is established, the other standby DSE instances connect to the live DSE instance and start synchronizing with it.
   - Select the DSE instance that you want to be the live DSE instance, and start that instance in ACTIVE state. Then start the other DSE instances on the other host systems in STANDBY state, in any order. The standby DSE instances then connect to the live DSE instance and start synchronizing with it.

Related tasks

“Enabling high availability storage systems for DSE” on page 99
Configure DSE to connect to a high availability storage system. Enabling high availability across DSE and the storage system provides the benefits of error recoverability and failover capability.

---

### High availability cluster management software

You must use management software to manage your high availability clusters.

This software performs the following functions:
- Monitors host systems and critical applications on host systems to ensure that they are responding
- Manages failover, so that when a critical component fails, other components can perform the functions of the failed component
- Supports critical services that are assigned to a specific service address, so that a service address moves to another host system when a failure occurs

Examples of high availability management software include:
- The Open Source High Availability Project (at www.linux-ha.org)

---

### High availability storage systems

You can connect a DSE high availability cluster to a storage system that is configured for high availability.
Enabling high availability across DSE and the storage system provides the following benefits:

- Failover capability and support for recovering and restarting failed components, as described in Chapter 14, “High availability clusters,” on page 95
- Error recovery capability, as described in Chapter 19, “Enabling error recovery for storage systems,” on page 131

**High availability DB2 configurations**

If you are using a DB2 database that has been configured for high availability, you can configure the peer DSE instances in a high availability cluster to connect to a single DB2 database, or you can connect each of the peer DSE instances to its own DB2 database.

The value of the HA_SHARED_DB parameter determines the configuration.

**Advantages of connecting the DSE instances to the same DB2 database**

- Less disk space is used. If each DSE instance is connected to its own DB2 database, the entire database is replicated for each DSE instance, and more disk space is used.
- The client can view the database as a single logical database that is highly available. If each DSE instance is connected to its own DB2 database, then the client must ensure that it is connected to the database that is connected to the live DSE instance. When failures occur, the live instance changes as a result of failover.
- The client does not need to keep track of which DSE instance is currently the live instance. (With multiple databases, each database is connected to its own DSE instance. If a failure occurs and the live DSE instance goes offline, the client needs to keep track of which DSE instance is currently the live instance.)
- You do not need to worry about synchronizing symbol IDs because you are using a single database, not multiple databases. If each DSE instance is connected to its own DB2 database, symbol IDs are not synchronized between the databases. When DSE inserts a new symbol into the database, the symbol is given a numeric symbol ID. Because the symbol IDs are not synchronized between the databases, the same symbol name can have different symbol IDs in each of the databases.

Connecting each DSE instance to its own DB2 database instance might be an easier configuration to set up. This configuration is useful when:

- You are only using DSE to save the symbols table to a store.
- Either new symbol names are not allowed, or symbol IDs are not used.

**DB2 database configuration**

A high availability DB2 database can be configured for High Availability Cluster Multi-Processing (HACMP™) or for high availability disaster recovery (HADR). A high availability DB2 database uses the DB2 automatic client reroute feature. The DB2 database configuration must meet the following requirements:

- The database is distributed over a cluster in multiple partitions.
- High availability cluster management software is installed on the cluster to provide failover and restart capabilities for the database servers.
- The tables for the DSE ticks and symbols are partitioned over the available partitions.
• Each partition has an assigned failover server. Use the DB2 command UPDATE ALTERNATE SERVER, or an equivalent mechanism.

In addition, the connection between DSE and the DB2 database must use TCP/IP.

### Enabling high availability storage systems for DSE

Configure DSE to connect to a high availability storage system. Enabling high availability across DSE and the storage system provides the benefits of error recoverability and failover capability.

To enable DSE to connect to high availability storage systems (such as the DB2 database system), complete the following steps:

1. Configure and start a DSE high availability cluster.
2. Set the HA_SHARED_DB parameter to indicate whether the peer DSE instances in the high availability cluster are connected to a single database, or each instance is connected to its own database instance.
3. Set the STORESYSXX_HA parameter to 1, to indicate that the storage system is enabled for high availability.
4. Set the following storage system configuration parameters to define how DSE retries failed operations and reconnects after losing a connection to the storage system:
   - STORESYSXX_RECONNECT_COUNT
   - STORESYSXX_RECONNECT_PERIOD
   - STORESYSXX_RETRY_COUNT
   - STORESYSXX_RETRY_PERIOD

**Related reference**

“Storage system configuration parameters” on page 71

DSE stores data in stores that are contained in a storage system. The storage system can be a relational database management system like a DB2 system, or it can be another kind of storage system. Use the storage system configuration parameters to configure the storage system.

“Configuring and starting a high availability cluster” on page 96

Configure a DSE high availability cluster to utilize the failover capability and prevent loss of data that result from system failures.
Chapter 15. Scenario: Building and managing a high availability cluster

This scenario demonstrates how to build and manage a high availability cluster on Linux. In this scenario, you install the software that is needed on each of the host systems, configure the software, and use IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM) to manage the cluster.

- This scenario requires five host systems that meet the minimum requirements to install the required software.
- You must have root authority to install the DSE wrapper and to configure IBM Tivoli SAM.

The high availability cluster includes two DSE instances and a DB2 HADR database (with a standby database). A simple data feed is simulated by the send_ticks program that is included with DSE. You use IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM) to manage this high availability cluster.

[Figure 2 on page 102](#) shows the five host systems that comprise the high availability cluster in this scenario. This figure shows the initial state of the cluster; failover can cause this configuration to change. For example, the primary DB2 HADR database can become the standby database, and the live DSE instance can become the standby instance.
Each of the two DSE instances runs on one host system. Together, these two systems comprise a simple high availability cluster, as described in Chapter 14, "High availability clusters," on page 95. In this scenario, one system hosts the live DSE instance, and the other system hosts the standby DSE instance. After a normal startup, the DSE_1 system hosts the live DSE instance and the DSE_2 system hosts the standby DSE instance. This cluster of DSE instances ensures continuous availability of data. If the live DSE instance fails, the standby instance takes over as the live instance and no data is lost.

This scenario also uses DB2 high availability disaster recovery (HADR), a DB2 database replication feature. HADR is a high availability solution that protects against data loss by replicating changes from a source database, called the primary database, to a target database, called the standby database. In this scenario, the primary HADR database runs on the DB2_1 system. The standby HADR database on the DB2_2 system is a backup copy of the primary HADR database.

IBM Tivoli SAM is an example of the high availability cluster management software that is described in "High availability cluster management software" on page 97.
In this scenario, IBM Tivoli SAM monitors the four systems that host the DSE instances and the DB2 HADR primary and standby databases. If a failure occurs, IBM Tivoli SAM manages failover so that the standby resource can take over the work of the failed resource.

This scenario uses the DSE send_ticks program to simulate a simple data feed. The send_ticks program runs on the FEED system and sends data to the DSE_1 and DSE_2 systems.

After the high availability cluster is configured and IBM Tivoli SAM is set up, you can query data from the DB2 HADR database on the DB2_1 system through the DSE wrapper.

Remember: Be sure to replace all of the names, directories, and parameter values in this scenario with information that is appropriate to your environment.

This scenario includes the following tasks:
1. “Installing the software components” on page 104
2. “Preparing IBM Tivoli SAM to manage the high availability cluster” on page 104
3. “Setting up the DB2 HADR databases” on page 105
4. “Setting up DSE” on page 106
5. “Testing DSE” on page 108
6. “Configuring IBM Tivoli SAM to manage the high availability cluster” on page 111
7. “Starting and testing the high availability cluster” on page 115
8. “Creating a DB2 wrapper and running queries” on page 118

Host systems for the high availability cluster scenario

This scenario was developed and tested with the five host systems running on Linux RHEL 3.0, Update 2 (32-bit). The five host systems in this scenario are named DSE_1, DSE_2, DB2_1, DB2_2, and FEED.

These systems must support the following software components of the high availability cluster:

**DSE_1**
- DSE server
- DB2 client (installed with DB2 Version 9.5 for Linux, UNIX, and Windows)
- IBM Tivoli SAM Version 2.1

**DSE_2**
- DSE server
- DB2 client (installed with DB2 Version 9.5 for Linux, UNIX, and Windows)
- IBM Tivoli SAM Version 2.1

**DB2_1**
- DB2 Version 9.5 for Linux, UNIX, and Windows
- IBM Tivoli SAM Version 2.1
- DSE wrapper
DB2_1 hosts the primary DB2 HADR database.

**DB2_2**
- DB2 Version 9.5 for Linux, UNIX, and Windows
- IBM Tivoli SAM Version 2.1
- DB2 Data Stream Engine wrapper

DB2_2 hosts the standby DB2 HADR database.

**FEED**
- DSE server

FEED hosts the send_ticks program that simulates a simple data feed.

You can have a simpler configuration that uses two host systems. The first system hosts the live DSE instance and the primary DB2 HADR database. The second system hosts the standby DSE instance and the standby DB2 HADR database. In this simpler configuration, IBM Tivoli SAM is installed on both systems and the send_ticks program runs on one of the systems.

### Installing the software components

To run this scenario, install the software components on each of the five host systems (DSE_1, DSE_2, DB2_1, DB2_2, and FEED).

To install the software:
1. Install and configure the DSE server on DSE_1, DSE_2, and FEED.
   a. Install the DSE server on DSE_1, DSE_2, and FEED. See Chapter 4, “Installing DSE,” on page 11 for instructions. Be sure to install the DSE server in the same file system location on DSE_1 and DSE_2. In this scenario, DSE is installed in the /local/dse directory on DSE_1, DSE_2, and FEED.
   b. Set up the DB2 environment for DSE on DSE_1 and DSE_2. See “Setting up a test storage system” on page 17 for instructions.
2. Install the DB2 database system on DSE_1 and DSE_2.
3. Install the DB2 database system on DB2_1 and DB2_2.
4. Install IBM Tivoli SAM Version 2.1 on DSE_1, DSE_2, DB2_1, and DB2_2. See the IBM Tivoli SAM documentation for information.

### Preparing IBM Tivoli SAM to manage the high availability cluster

In this scenario, the five host systems, or nodes, comprise the high availability cluster. Preparing IBM Tivoli SAM to manage the high availability cluster involves preparing the nodes in the cluster and creating the cluster domain.

You must run the commands in this section as root.

To prepare IBM Tivoli SAM:
1. Prepare each of the nodes of the cluster by issuing the following commands on each host system:
   ```bash
   setenv CT_MANAGEMENT_SCOPE 2
   /usr/sbin/rsct/bin/preprpnode db2_1 db2_2 dse_1 dse_2
   ```
2. On any one of the systems, create the cluster domain by issuing the following command:
Setting up the DB2 HADR databases

This scenario involves two DB2 HADR databases that act as the live and standby databases.

Set up the DB2 HADR database on DB2_1 and the standby database on DB2_2.

You can also automate the high availability management of the DB2 HADR database using IBM Tivoli SAM. For instructions, see the white paper titled Automating DB2 HADR Failover on Linux using Tivoli System Automation at ftp://ftp.software.ibm.com/software/data/db2/linux/tsa_hadr.pdf.

To set up the DB2 HADR database on DB2_1 and the standby database on DB2_2:

1. Create the DB2 instances.
   a. On DB2_1, create a DB2 instance. For this scenario we will name the instance db2inst2.
   b. On DB2_2, create a DB2 instance. For this scenario we will name the instance db2inst3.

   Ensure that the names of the DB2 instances are not already in use. The names of the DB2 instances must be unique across host systems.


2. Install the DSE wrapper on DB2_1 and on DB2_2. You must log on as root to install the wrapper. See Chapter 4, “Installing DSE,” on page 11 for instructions.

3. Create the primary DB2 HADR database, named hadrdb, on DB2_1. Log in as the db2inst2 user and run the following command:
   
   ```
   db2 create database hadrdb
   ```

   This step might take some time to complete.
4. Create the ticks and symbols tables in the primary DB2 HADR database on DB2_1.
   a. On DB2_1, create a copy of the examples/ha/db2-schema.sql file and put the copy of the file in the working directory. Ensure that the copy of the file is writable.
   b. Edit the copy of the db2-schema.sql file and remove the CONNECT statement.
   c. Connect to the DB2 HADR database on DB2_1 by issuing the following command on DB2_1:
      ```
      db2 connect to hadrdb user db2inst2
      ```
   d. Create the ticks and symbols table in that database by issuing the following command on DB2_1:
      ```
      db2 -tf db2-schema.sql
      ```
   e. On DB2_1, set up database configuration by issuing the following commands:
      ```
      db2 update db cfg for hadrdb using LOGRETAIN ON
      db2 update db cfg for hadrdb using LOGINDEXBUILD ON
      db2 update db cfg for hadrdb using INDEXRESTART
      db2 backup database hadrdb
      ```
      These commands create a database backup file with a name similar to the following name:
      ```
      HADRDB.0.db2inst2.NODEN0000.CATN0000.20060201155839.001
      ```

5. Create a standby DB2 HADR database on DB2_2.
   a. Log onto DB2_2 as the db2inst3 user.
   b. Copy the database backup file into the home directory of the db2inst3 user and restore the database by issuing the following command:
      ```
      db2 restore database testhadr
      ```

   **Note:** If you have more than one backup file in the home directory, then you need to do one of the following:
   - Specify which backup file you want to restore.
   - Remove all backup files except the one that you plan to restore.

You need to complete additional steps to complete the setup of the HADR database and automate its management with IBM Tivoli SAM. See the white paper titled *Automating DB2 HADR Failover on Linux using Tivoli System Automation* at ftp://ftp.software.ibm.com/software/data/db2/linux/tsa_hadr.pdf.

### Setting up DSE

Set up DSE on the DSE_1 and DSE_2 systems to become parts of the high availability cluster in this scenario. After setting up DSE on these systems, you set up the connection between these systems and the DB2_1 and DB2_2 systems. The steps in this section are similar to the steps to verify the DSE installation.

To set up DSE:

1. Modify the DSE configuration files on DSE_1 and DSE_2. On each system, do the following steps:
   a. Create a copy of the rtlconfig configuration file that is in the examples/ha/config subdirectory, and put the copy of the file in the installation directory (the parent directory of the examples subdirectory). Ensure that the copy of the configuration file is writable.
b. Modify the appropriate parameters in the copy of the configuration file. The following sample shows the configuration parameters that you should consider changing from their default values. Replace the configuration parameter values with your own values.

```plaintext
RTL_SERVICE_NAME "18890"
RTL_SYS_MSG_PATH "/local/dse/msg/rltysmmsgs.txt"
RTL_LOG_PATH "/local/dse/logs/rtl.log"
HA01_PEER_INFO "dse_1.svl.ibm.com:18888"
HA02_PEER_INFO "dse_2.svl.ibm.com:18889"
HA01_PEER_NAME "dse_1"
HA02_PEER_NAME "dse_2"
HANDLER01_MSG_PATH "/local/dse/examples/ha/handler/hahandlermsgs.txt"
HA_SYNC_DELAY 3
HA_RETRY_DELAY 3
HA_LIB_PATH "/local/dse/lib/hasys.so"
HA_MSG_PATH "/local/dse/msg/rtlhamsgs.txt"
HA_SHARED_DB 1
STORESYS01_DB 1
STORESYS01_CONN "hadrdb" # name of database
STORESYS01_USER "db2inst2" # database user
STORESYS01_PASSWORD "cac08f678ec2bd6c67376a75797c4763"
STORESYS01_LIB_PATH "/local/dse/lib/ha"
STORESYS01_MSG_PATH "/local/dse/msg/rtlstoremsgs-db2.txt"
STORE01_NAME "rtl.ticks" # match schema and table name in database
HANDLER01_NAME "/local/dse/lib/ha"
HANDLER01_LIB_PATH "/local/dse/lib/hahandler.so"
GROUP01_NAME "rtltest" # name of this group
GROUP01_PASSWORD "c9cfc29ebcc8c2e67c7a7c7b797c7463"
# password for this group (marshmello)
GROUP01_AUTHLEVEL 4 # authorization for this level
FEED01_STORE_NAME "rtl.ticks" # match schema and table name in database
FEED01_SOURCE "192.168.1.12:18878" # feed (on DSE_1)
FEED01_SOURCE "192.168.1.12:18879" # feed (on DSE_2)
```

Restrictions for the FEED01_SOURCE parameter:

- The values of the FEED01_SOURCE parameters for DSE_1 and DSE_2 must not be the same.
- Include only one FEED01_SOURCE parameter in the configuration file. This sample shows two FEED01_SOURCE parameter definitions to show the sample values for DSE_1 and DSE_2.
- The FEED01_SOURCE parameter value consists of two parts: the IP address and the port number. For example, in the FEED01_SOURCE parameter value for DSE_1, the IP address is 192.168.1.12 and the port number is 18878. Be sure to replace these values with the IP address and port numbers of your systems.

Remove the SHMKEY parameter if you want to use the default value.

See Chapter 9, “System configuration parameters,” on page 51 for information about these parameters.

2. Build the high availability feed handler on DSE_1 and on DSE_2. On each system, do the following steps:
   a. Change directories to the examples/ha/handler directory.
   b. Run the make command to build the high availability feed handler.
   c. Copy the hahandlermsgs.txt file to the msg directory. The high availability feed handler uses the hahandlermsgs.txt file as a message file, but this file is not copied to that directory automatically.

   Run the following commands:
   ```bash
   cd ../../..
   cp examples/ha/handler/hahandlermsgs.txt msg
   ```
3. Connect DSE on DSE_1 and DSE_2 to the DB2 HADR database on DB2_1. This connection extends to the backup database on DB2_2. Complete these steps on both DSE_1 and DSE_2, as the DB2 db2inst2 user. The connection values in this step correspond to the values of the STORESYS01_CONN, STORESYS01_USER, and STORESYS01_PASSWORD configuration parameters in the configuration file (see step [1b on page 107].

   a. Run the following commands:

   ```
   db2 catalog tcpip node db2_1 remote 192.168.1.10 server 60000
   db2 catalog database hadrdb at node db2_1
   ```

   In these commands:

   - 192.168.1.10 is the IP address of the remote DB2_1 system.
   - 60000 corresponds to the value in the /etc/services file for the TCP/IP connection that is used by the DB2 instance db2inst2. The following part of a sample services file contains this information:

     ```
     DB2_db2inst2 60000/tcp
     DB2_db2inst2_1 60001/tcp
     DB2_db2inst2_2 60002/tcp
     DB2_db2inst2_END 60003/tcp
     DB2_db2inst3 60010/tcp
     DB2_db2inst3_1 60011/tcp
     DB2_db2inst3_2 60012/tcp
     DB2_db2inst3_END 60013/tcp
     ```

   b. Test the connection by issuing the following command: On DSE_1:

   ```
   db2 connect to hadrdb user db2inst2
   ```

   On DSE_2:

   ```
   db2 connect to hadrdb user db2inst2
   ```

   After you enter the password, you should see output that is similar to the following output:

   ```
   Database Connection Information
   Database server = DB2/LINUX 8.2.3
   SQL authorization ID = db2inst2
   Local database alias = HADRDB
   ```

---

**Testing DSE**

Run the send_ticks program on the FEED system to simulate a simple data feed. Then determine that DSE on DSE_1 and DSE_2 received the ticks, to test that DSE is working properly on those systems.

To test DSE:

1. Configure the send_ticks program on the FEED system. On FEED:
   a. Change directories to the examples/ha/driver directory.
   b. Run the make command.

2. Define the port numbers of DSE_1 and DSE_2 to FEED, by creating a file named dest.cfg in the driver subdirectory on FEED. Include the following contents in this file:

   ```
   18878
   18879
   ```
In this file, the first line represents DSE_1 and the second line represents DSE_2.

- 18878 is the port number of DSE_1.
- 18879 is the port number of DSE_2.

Each line must match the corresponding FEED01_SOURCE parameter value in the DSE configuration file. See “Setting up DSE” on page 106.

3. Test that DSE on DSE_1 is working properly by running the send_ticks program on FEED.
   a. Edit the dest_cfg file on FEED and remove the # character from the first line, so that FEED can read the port to send ticks to the DSE feed handler on DSE_1. The contents of the dest_cfg file should be as follows:
      18878
      18879
   b. Start the send_ticks program on FEED and send the ticks to DSE_1 by issuing the following command on FEED:
      send_ticks -ftick_data -sdest_cfg
      The send_ticks program automatically exits after it connects to the feed handler and sends the 1000 ticks that are in the tick_data file.
   c. Start DSE on DSE_1 with your copy of the configuration file, by issuing the following command:
      rtloader -c rtlconfig &
   d. After the send_ticks program exits, check the rtl.log log file in the logs subdirectory to ensure that DSE on DSE_1 started successfully and that it received the ticks from the send_ticks program. The following sample log file shows that the test is successful. The results that you get should be similar to this output:

```plaintext
[2006-02-01T16:48:18-07:00] DSEI00001: Data Stream Engine
[2006-02-01T16:48:18-07:00] DSEI01921: Execution begins
[2006-02-01T16:48:18-07:00] DSEI01631: Allocated room for 10000 symbols
[2006-02-01T16:48:18-07:00] DSEI00841: 230890 ticks initialized
[2006-02-01T16:48:18-07:00] DSEF00051: Handler 1 (ha) Initialised
[2006-02-01T16:48:18-07:00] DSES00011: Storage System db2 initialized
[2006-02-01T16:48:18-07:00] DSEI01581: Initialized 0 Symbols for Store 1
[2006-02-01T16:48:18-07:00] DSEI00521: Loaded 0 symbols into memory
[2006-02-01T16:48:18-07:00] DSEI01561: Starting New Symbol thread 47950768
[2006-02-01T16:48:18-07:00] DSEI01571: Starting Purge thread 112618416
[2006-02-01T16:48:18-07:00] DSEI01581: Starting Feed Handler thread 58440624
[2006-02-01T16:48:18-07:00] DSEF05091: HA Feed handler initialized
[2006-02-01T16:48:18-07:00] DSEI01591: Starting Listener thread 68930480
[2006-02-01T16:48:18-07:00] DSEI00031: Initialized feed 1 with stamps 2001020514321390000 through 2001020514322300000
[2006-02-01T16:48:18-07:00] DSEI01601: Starting Main Query thread 79420336
[2006-02-01T16:48:18-07:00] DSEI02171: Starting HA Listener thread 133598128
[2006-02-01T16:48:18-07:00] DSEI00411: Feed thread 58440624 exited
```
e. Shut down DSE by issuing the following command on DSE_1:
   ```bash
   rtmlmode -ky
   ```

f. Clean up the DB2 HADR database by issuing the following commands on DSE_1:
   ```bash
   db2 connect to hadr db2inst2
   db2 delete from rtl_ticks_ticks
   db2 delete from rtl_ticks_symbols
   ```

g. Remove the rtl.log file by issuing the following command on DSE_1:
   ```bash
   rm logs/rtl.log
   ```

4. Test that DSE on DSE_2 is working properly by running the send_ticks program on FEED.
   a. Edit the dest_cfg file on FEED. Add the # character to the beginning of the first line. Remove the # character from the second line, so that FEED can read the port to send ticks to the DSE feed handler on DSE_2. The contents of the dest_cfg file should be as follows:
   ```bash
   18878
   18879
   ```
   b. Repeat steps [3b on page 109](#) through [3e](#) for DSE_2 instead of DSE_1.

5. Test that the DSE instances on both DSE_1 and DSE_2 work simultaneously as a high availability cluster. Run the send_ticks program on FEED and send the ticks to both DSE_1 and DSE_2 simultaneously.
   a. Edit the dest_cfg file on FEED and remove the # character from the first line. Ensure that neither of the lines in the file contain the # character. The contents of the dest_cfg file should be as follows:
   ```bash
   18878
   18879
   ```
   b. Start the send_ticks program on FEED and send the ticks to both DSE_1 and DSE_2 by issuing the following command on FEED:
   ```bash
   send_ticks -Tn -I100 -ftick_data -sdest_cfg
   ```
   The -Tn parameter is required when the send_ticks program is sending data to more than one target. When you use this parameter, ticks are sent in batches of 100 with a delay of n seconds between each batch.
   The -I parameter specifies an initial delay of n seconds before the send_ticks program starts sending ticks. Using this parameter enables you to start DSE on both DSE_1 and DSE_2 before the send_ticks programs starts sending ticks. The send_ticks program automatically exits after it connects to the feed handler and sends the 1000 ticks that are in the tick_data file.
   c. On DSE_1, start DSE in live mode by issuing the following command:
   ```bash
   rtlloader -c rtlconfig -M1 &
   ```
   Then complete the next step within the initial delay period of the send_ticks program.
   d. On DSE_2, start DSE in standby mode by issuing the following command:
   ```bash
   rtlloader -c rtlconfig -M5 &
   ```
   e. Check the log files on both DSE_1 and DSE_2 to ensure that DSE started successfully on both machines.
   f. Clean up the DB2 HADR database by issuing the following commands on DSE_1:
   ```bash
   db2 connect to hadr db2inst2
   db2 delete from rtl_ticks_ticks
   db2 delete from rtl_ticks_symbols
   ```
g. Remove the rtl.log file from both DSE_1 and DSE_2, by issuing the following command on each system:

```
rm logs/rtl.log
```

---

**Configuring IBM Tivoli SAM to manage the high availability cluster**

IBM Tivoli SAM is the cluster management software that is used in this scenario. Configure this software to manage the high availability cluster and automate failover and recovery.

This step creates a floating IP address for the live DSE instance that is either on the DSE_1 system or the DSE_2 system. The DSE wrapper accesses the live DSE instance using this floating IP address. IBM Tivoli SAM binds the floating IP address to whichever host system has the live DSE instance, and moves the IP address when the live DSE instance moves to the other host system.

The commands in this section are IBM Tivoli SAM commands.

To configure IBM Tivoli SAM to manage the high availability cluster:

1. Set up the management script that is included with DSE to configure DSE for IBM Tivoli SAM. On DSE_1 and on DSE_2, do the following steps:
   a. Create a copy of the management script file and put it in the bin directory, by issuing the following command in the bin directory:
   ```
cp /local/dse/examples/ha/setup/tsa_manage.ksh /local/dse/bin
   ```
   b. Ensure that the copy of the script file is writable. Then change the following parameters in the file:

   **CODEHOME**
   Change the value of this parameter to the full path of the installation bin directory.

   **CONFIG**
   Change the value of this parameter to the full path of the DSE configuration file.

   **INSTNAME**
   Change the value of this parameter to the name of the DB2 instance that you are using.

   **SHMKEY**
   Change the value of this parameter if you are not using the default value.

   For example:
   ```
   CODEHOME=/local/dse/bin
   CONFIG=/local/dse/rtlconfig
   INSTANCE=db2inst2
   # Change the shared memory key from the default here
   SHMKEY=268658176
   ```
   c. Test the script file by issuing the following command:
   ```
   tsa_manage.ksh test
   ```
   You should see output that is similar to the following output:
1. Run the following commands:
   - `/local/dse/bin/rtloader`
   - `/local/dse/bin/rtmode`
   - `/local/dse/bin/rtlstat`
   - `/qa/local/db2inst2/sqlib/db2profile`
   - `/local/dse/rtlconfig`
   - `/local/dse/bin/rtlmode`

   The values in the output should match the values that you updated in step 1b on page 111.

   You should not see any errors.

2. Create the network resource. The network resource groups the DSE_1 and DSE_2 systems together, and has its own IP address. IBM Tivoli SAM references this floating IP address, so that it does not need to know whether the live DSE instance is on DSE_1 or DSE_2.

   Complete the remaining steps in this section as root.

   a. On any one of the systems, run the following command:
      ```bash
      setenv CT_MANAGEMENT_SCOPE 2
      ```

   b. Identify the names of the Ethernet adaptors of DSE_1 and DSE_2 by issuing the following command on any one of the systems:
      ```bash
      lsrsrc IBM.NetworkInterface Name IPAddress
      ```

      You should see output that is similar to the following output:

      ```
      Resource Persistent Attributes for IBM.NetworkInterface
      resource 1:
        Name = "eth0"
        IPAddress = "192.168.1.10"
      resource 2:
        Name = "eth0"
        IPAddress = "192.168.1.11"
      resource 3:
        Name = "eth0"
        IPAddress = "192.168.1.12"
      resource 4:
        Name = "eth0"
        IPAddress = "192.168.1.14"
      resource 5:
        Name = "eth0"
        IPAddress = "192.168.1.15"
      ```

      The output of this command shows the Ethernet adaptor names and IP addresses for each of the nodes. If you see more than one Ethernet adaptor name for each node, choose one name. Use the names of the DSE_1 and DSE_2 Ethernet adaptors in the mkequ command in the next step.

   c. On any one of the systems, make a resource equivalency for DSE_1 and DSE_2 by issuing the following commands:
      ```bash
      mkequ dse_eth_equ IBM.NetworkInterface:eth0:dse_1,eth0:dse_2
      mkrsrc IBM.ServiceIP Name="dse_sip_rs" \
        IPAddress="192.168.1.81" \
        NetMask="255.255.255.0" \
        NodeNameList="{'dse_1','dse_2'}"
      ```

      In these commands:
      - Allocate an unused IP address on your network for the floating IP address. This scenario uses 9.30.187.81 as the floating IP address. This IP address cannot be used by any of the nodes.
      - The eth0 name in the mkequ command is the name of the Ethernet adaptor for both DSE_1 and DSE_2.
      - The mkrsrc command creates the resource. In this scenario, the name of the resource is dse_sip_rs.
3. Create three DSE resources:
   
   • One DSE resource on DSE_1, to be started as a standby resource.
   • One DSE resource on DSE_2, to be started as a standby resource.
   • One floating DSE resource, to be started as the live resource. This floating resource maps to the actual live resource that can be on DSE_1 or DSE_2.

Complete the following steps on any one of the systems.

a. Create the DSE resource on DSE_1 by issuing the following command:

   ```
   mkrsrc IBM.Application Name="dse_dse_1_rs" 
   ResourceType=0 
   StartCommand="/local/dse/bin/tsa_manage.ksh start" 
   StopCommand="/local/dse/bin/tsa_manage.ksh stop" 
   MonitorCommand="/local/dse/bin/tsa_manage.ksh status" 
   StartCommandTimeout=20 
   StopCommandTimeout=20 
   UserName=rjt 
   NodeNameList="{'dse_1'}"
   ```

   In this command:
   • The name of the DSE resource on DSE_1 is dse_dse_1_rs. Replace this name with one of your choosing, if necessary.
   • In the StartCommand, StopCommand, and MonitorCommand statements, replace the location of the management script file with the actual location of the file on your system.
   • The timeout values of 20 are larger than the default, to allow time for synchronization between the DSE resources.
   • Replace the value of the UserName parameter with your own user name.
   • Replace the value of the NodeNameList parameter with the name of your node.

b. Create the DSE resource on DSE_2 by issuing the following command:

   ```
   mkrsrc IBM.Application Name="dse_dse_2_rs" 
   ResourceType=0 
   StartCommand="/local/dse/bin/tsa_manage.ksh start" 
   StopCommand="/local/dse/bin/tsa_manage.ksh stop" 
   MonitorCommand="/local/dse/bin/tsa_manage.ksh status" 
   StartCommandTimeout=20 
   StopCommandTimeout=20 
   UserName=rjt 
   NodeNameList="{'dse_2'}"
   ```

   In this command:
   • The name of the DSE resource on DSE_2 is dse_dse_2_rs. Replace this name with one of your choosing, if necessary.
   • In the StartCommand, StopCommand, and MonitorCommand statements, replace the location of the management script file with the actual location of the file on your system.
   • The timeout values of 20 are larger than the default, to allow time for synchronization between the DSE resources.
   • Replace the value of the UserName parameter with your own user name.
   • Replace the value of the NodeNameList parameter with the name of your node.

c. Create a resource group for the standby DSE resources on DSE_1 and DSE_2 and add the resources to that group.

   1) Create the resource group by issuing the following command:

   ```
   mkrg -l None dse_rg
   ```
The name of this resource group is dse_rg. The -I None option in this command indicates that this resource group is not linked to a specific node.

2) Add the DSE resources on DSE_1 and DSE_2 to the dse_rg resource group by issuing the following commands:
   
   addrgmbr -g dse_rg IBM.Application:dse_dse_1_rs:dse_1
   addrgmbr -g dse_rg IBM.Application:dse_dse_2_rs:dse_2

3) Ensure that the resource group does not fail if one of its members goes offline, by issuing the following commands:
   
   chrgmbr -m F -g dse_rg IBM.Application:dse_dse_1_rs:dse_1
   chrgmbr -m F -g dse_rg IBM.Application:dse_dse_2_rs:dse_2

   In these commands, replace the name of the resource group (dse_rg), DSE resources (dse_dse_1_rs and dse_dse_2_rs), and node names with your own names.

4) Create a floating DSE resource that maps to the live DSE instance on either DSE_1 or DSE_2. Run the following command:
   
   mkrsrc IBM.Application Name="dse_live_rs" \ 
   ResourceType=1 \ 
   StartCommand="/local/dse/bin/tsa_manage.ksh livestart" \ 
   StopCommand="/local/dse/bin/tsa_manage.ksh livestop" \ 
   MonitorCommand="/local/dse/bin/tsa_manage.ksh livestatus" \ 
   StartCommandTimeout=5 \ 
   StopCommandTimeout=20 \ 
   UserName=rjt \ 
   NodeNameList="{dse_1,'dse_2'}"

   In this command:
   • The name of the floating DSE resource is dse_live_rs. Replace this name with one of your choosing, if necessary.
   • In the StartCommand, StopCommand, and MonitorCommand statements, replace the location of the management script file with the actual location of the file on your system.
   • Replace the value of the UserName parameter with your own user name.
   • Replace the value of the NodeNameList parameter with the names of your nodes, if they are not named DSE_1 and DSE_2.

5) Create a resource group for the floating DSE resource. Then add the network resource (dse_sip_rs) and the floating DSE resource (dse_live_rs) to this resource group. Run the following commands:
   
   mkrgr 1 None dse_svc_rg
   addrgmbr -g dse_svc_rg IBM.ServiceIP:dse_sip_rs
   addrgmbr -g dse_svc_rg IBM.Application:dse_live_rs

   In these commands, the name of the resource group is dse_svc_rg. Replace this name with one of your choosing, if necessary.

4. Create managed relationships between the resources that you created. Complete the following steps on any one of the systems.

   a. Create a relationship between the network resource for DSE_1 and DSE_2 (dse_sip_rs) and the equivalent Ethernet component. Run the following command:
      
      mkrrel -p DependsOn \ 
      -S IBM.ServiceIP:dse_sip_rs \ 
      -G IBM.Equivalency:dse_eth_equ \ 
      -G IBM.Equivalency:dse_eth_dprel

   b. Create a relationship that indicates that the live DSE resource must be on the same node as the network resource. Run the following command:
c. Link the live DSE resource to the actual underlying DSE instances on DSE_1 and DSE_2.
   1) Create a relationship that indicates that the DSE instance on DSE_1 cannot be the live instance if DSE_1 is not online, by issuing the following command:
      mkrel -p AntiAffinity -o IfNotOnline
          -S IBM.Application:dse_live_rs
          -G IBM.Application:dse_dse_1_rs:dse_1
          dse_live_gg_afrel
   2) Create a relationship that indicates that the DSE instance on DSE_2 cannot be the live instance if DSE_2 is not online, by issuing the following command:
      mkrel -p AntiAffinity -o IfNotOnline
          -S IBM.Application:dse_live_rs
          -G IBM.Application:dse_dse_2_rs:dse_2
          dse_live_ip_afrel

Starting and testing the high availability cluster

After the high availability cluster is set up, start and test it to ensure that it is working properly in this scenario.

To start and test the high availability cluster:

1. Set the nominal state of the DSE resource groups (dse_rg and dse_svc_rg) to online.
   a. Ensure that these resources do not start prematurely by issuing the following IBM Tivoli SAM commands as root on any one of the systems:
      rgreq -o stop dse_rg
      rgreq -o stop dse_svc_rg
   b. Set the nominal state of the resources groups to online by issuing the following IBM Tivoli SAM commands as root on any one of the systems:
      chrg -o Online dse_rg
      chrg -o Online dse_svc_rg

2. Start the high availability cluster.
   a. On FEED, start the send_ticks program by issuing the following command:
      send_ticks -T1 -I10 -ftick_data -s dest_cfg
   b. Start the DSE resources on DSE_1 and DSE_2. On any one of the systems, run the following command:
      rgreq -o start dse_rg
   c. Start the live DSE resource group (which maps to the live DSE instance) by issuing the following command on any one of the systems:
      rgreq -o start dse_svc_rg

3. Check the status of the configuration with the DB2 getstatus command. As root, run the following command on any one of the systems:
   /opt/ibm/db2/V9.1/ha/lsa/getstatus

   This command displays the existing resource groups, the resources in those groups, and the status of each resource. The following sample output is from this command:
This sample output shows that the network resource (dse_sip_rs) and the floating DSE resource (dse_live_rs) are online on DSE_1. Thus, these resources are offline on DSE_2.

4. Test moving the live DSE instance from the online system to the offline system. For example, if the online system is DSE_1, move the live DSE instance from DSE_1 to DSE_2 by issuing the following command on any one of the systems:
   
   \texttt{rgreq -o move -n dse_1 dse_svc霰g}

5. Check the status of configuration again, to confirm that DSE_2 is now online, and therefore the live DSE instance is on DSE_2. Run the following command on any one of the systems:

   \texttt{/opt/ibm/db2/V9.1/ha/tsa/getstatus}

The output should show that DSE_1 is offline and DSE_2 is online, as follows:

--- Resource Groups and Resources ---

<table>
<thead>
<tr>
<th>Group Name</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>dse_rg</td>
<td>dse_dse_2_rs</td>
</tr>
<tr>
<td>dse_rg</td>
<td>dse_dse_1_rs</td>
</tr>
<tr>
<td>dse_svc_rg</td>
<td>dse_live_rs</td>
</tr>
<tr>
<td>dse_svc_rg</td>
<td>dse_sip_rs</td>
</tr>
</tbody>
</table>

--- Resources ---

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Node Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>dse_dse_2_rs</td>
<td>dse_2</td>
<td>Online</td>
</tr>
<tr>
<td>dse_dse_1_rs</td>
<td>dse_1</td>
<td>Online</td>
</tr>
<tr>
<td>dse_live_rs</td>
<td>dse_1</td>
<td>Online</td>
</tr>
<tr>
<td>dse_live_rs</td>
<td>dse_2</td>
<td>Offline</td>
</tr>
<tr>
<td>dse_sip_rs</td>
<td>dse_1</td>
<td>Online</td>
</tr>
<tr>
<td>dse_sip_rs</td>
<td>dse_2</td>
<td>Offline</td>
</tr>
</tbody>
</table>
Testing the high availability cluster is complete.

6. On each of the DSE_1 and DSE_2 systems, run the rtlstat -m command to show the number of symbols that are in shared memory. The live and standby DSE instances should have the same number of symbols in shared memory:
   • After the DSE instances on these systems synchronize.
   • If the dse_svc_rg resource group was started before the send_ticks program started sending ticks.

The following sample output is from issuing the rtlstat -m command on DSE_1:

```
rtloader Version 9.1.0.0 -- Live -- On-Line -- Up 00:00:51 -- 0 Kbytes
```

<table>
<thead>
<tr>
<th>Ticks</th>
<th>Ticks</th>
<th>Ticks</th>
<th>Ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loaded</td>
<td>Free</td>
<td>Purged</td>
<td>In Shared Memory</td>
</tr>
<tr>
<td>1000</td>
<td>229890</td>
<td>0</td>
<td>1000</td>
</tr>
</tbody>
</table>

The following sample output is from issuing the rtlstat -m command on DSE_2:

```
rtloader Version 9.1.0.0 -- Standby -- On-Line -- Up 00:03:49 -- 0 Kbytes
```

```
Shared Memory Size : 16777216
Total ticks in pool : 230890 Used : 1000 [0%] Highest [2%]
```

7. Stop the live DSE resource group by issuing the following command on any one of the systems:
   
   `rgreq -o stop dse_svc_rg`

8. Stop the resource group for the standby DSE resources on DSE_1 and DSE_2 by issuing the following command on any one of the systems:
   
   `rgreq -o stop dse_rg`

9. Clean up the DB2 HADR database on DB2_1 by issuing the following commands on any one of the systems:
   
   `db2 connect to hadrdb user db2inst2`
   `db2 delete from rtl.ticks_ticks`
   `db2 delete from rtl.ticks_symbols`

10. On DSE_1 and on DSE_2, remove the rtl.log file by issuing the following command on each system:
    
    `rm logs/rtl.log`
Creating a DB2 wrapper and running queries

The final step in confirming that the high availability cluster works in this scenario is to demonstrate that you can query data from the DB2 HADR database on DB2_1 through the DSE wrapper for the DB2 database system.

To create a DB2 wrapper and run queries:

1. On any one of the systems, change directories to your working directory. Create a copy of the db2_sql/examples/ha/wrapper_schema.sql file and ensure that the copy of the file is writable.

2. Edit the copy of the wrapper_schema.sql file by doing the following steps:
   a. In Step 2 in the wrapper_schema.sql file, replace the value of the host name with the floating IP address for the network resource that you defined in “Configuring IBM Tivoli SAM to manage the high availability cluster” on page 111. The following example shows Step 2 in the wrapper_schema.sql file:

      ```sql
      echo === Step 2: create server ================;

      create server DSETEST wrapper DSE
      options (host_name '192.168.1.81', service_name '18890');
      The service_name value '18890' must match the value of the
      RTL_SERVICE_NAME parameter in the configuration file.
      ``

   b. In Step 3 of the wrapper_schema.sql file, change the db2inst1 name to the name of your DB2 instance (for example, db2inst2). The following example shows Step 3 in the wrapper_schema.sql file:

      ```sql
      echo === Step 3: create user mapping -- YOU MUST EDIT! ============;

      create user mapping for db2inst2 server DSETEST
      options (remote_authid 'rtltest', remote_password 'marshmello');
      The group name ('rtltest') must match the value of the GROUP01_NAME parameter in the configuration file. The password ('marshmello') must correspond to the value of the GROUP01_PASSWORD parameter in the configuration file.
      ```

3. On the DB2_1 system, connect to the DB2 HADR database hadrdb by issuing the following command:

   ```sql
   db2 connect to hadrdb user db2inst2
   ```

4. Start the high availability cluster.
   a. On FEED, start the send_ticks program by issuing the following command:

      ```
      send_ticks -T1 -I10 -ftick_data -sdest_cfg
      ```

   b. Start the DSE resources on DSE_1 and DSE_2. On any one of the systems, run the following command as root:

      ```
      rgreq -o start dse_rg
      ```

   c. Start the live DSE resource group (which maps to the live DSE instance) by issuing the following command on any one of the systems as root:

      ```
      rgreq -o start dse_svc_rg
      ```

5. Create the DB2 wrapper by issuing the following command on any one of the systems:

   ```
   db2 -tf wrapper_schema.sql
   ```

   If the command is successful, the four select query statements in Step 8 and Step 9 of the wrapper_schema.sql ran successfully.

6. Run queries through the wrapper.
a. Run some queries to read data from the cache of the live DSE instance, by issuing the following commands:

```
db2 -tv
db2 => select * from tick_data where symbol='652';
db2 => select * from tick_data where symbol='2194';
db2 => select * from meta_data where symbol='652';
db2 => select * from meta_data where symbol='2194';
```

The output should provide the requested data.

b. Move the live DSE instance from the online system to the offline system. For example, if the online system is currently DSE_2, move the live DSE instance from DSE_2 to DSE_1 by issuing the following command on any one of the systems as root:

```
rgreq -o move -n dse_2 dse_svc_rg
```

c. Run the same queries in step 6a. These queries request data from the cache of the new live DSE instance. The output should contain the same results that you saw in step 6a.

Getting the same results demonstrates that you successfully queried the data from both DSE instances on DSE_1 and DSE_2.

---

**Resource failures**

Sometimes, resources fail and you must manually fix the problem. Failures can result when the target database is not online, and DSE fails to start.

The DB2 getstatus command (described in "Starting and testing the high availability cluster" on page 115) shows resource failures. After you manually fix the failure, you can reset the resource by issuing the following command:

```
resetrsrc -s 'Name == "name"' resource class
```
Part 4. Administering storage systems

DSE works with the DB2 storage system to persist data from data streams and enable queries of that data. Storage system administration tasks include configuring the DB2 system, setting up tables, partitioning with DPF, recovering from errors or bad data, and enabling SQL queries.
Chapter 16. Configuration parameters for setting up the DB2 storage system

A number of DSE configuration parameters have a specific meaning for the DB2 storage system that is provided by DSE.

The following configuration parameters are required by the DB2 storage system. Additional configuration parameters that are not listed might also be required, but not specifically for the DB2 storage system.

**STORESYSXX_CONN**
- Specifies the DB2 database name that DSE connects to and uses for persistent storage.

**STORESYSXX_INFO**
- This parameter value is a formatted string that the DB2 storage system uses to specify whether the database is a multi-partitioned database that uses the DB2 massively parallel processing (MPP) database architecture.
  - If the value of STORESYSXX_INFO is MPP=1, DSE connects to each partitioning server directly to provide better performance when it flushes ticks to the persistent store.

**STORESYSXX_LIB_PATH**
- Specifies the path to the shared library. The DB2 storage system library file is db2relstoresys.so in the /lib subdirectory of the DSE installation.

**STORESYSXX_MSG_PATH**
- Specifies the path of the message file. The DB2 storage system message file, rt1storemsgs-db2.txt, is in the /msg subdirectory of the DSE installation.

**STORESYSXX_PASSWORD**
- Specifies the user password that is used by DSE for its database connections. This password is an encrypted string that you must create with the rtl_make_pass utility.

**STORESYSXX_USER**
- Specifies the user name that is used by DSE for its database connections. The user and password values can be left blank if you are connecting to the database as the user invoking DSE.

**STOREXX_FLUSH_BATCH**
- The batch size refers to the number of ticks that are written to the database in a single transaction.

**STOREXX_META_BATCH**
- The batch size refers to the number of symbols whose metadata changes are written to the database in a single transaction.

**STOREXX_SCHEMA**
- Specifies the schema name for the symbols table and the ticks table.

**STOREXX_SCHEMA_TYPE**
- Specifies whether the store uses a master-detail relationship or a simple schema.
STOREXX_SYMBOL_BATCH
The batch size refers to the number of new symbols that are written to the database in a single transaction.

STOREXX_SYMBOL_TABLE
Specifies the name of the symbols table.

STOREXX_TICK_TABLE
Specifies the name of the ticks table.

Related reference
“Storage system configuration parameters” on page 71
DSE stores data in stores that are contained in a storage system. The storage system can be a relational database management system like a DB2 system, or it can be another kind of storage system. Use the storage system configuration parameters to configure the storage system.

“Store configuration parameters” on page 74
DSE stores ticks and symbols from data feeds. For each store, you can set the store configuration parameters to customize data storage.
Chapter 17. DB2 database table schema

To use DSE with the DB2 storage system, you must set up a DB2 database table or tables.

DSE supports stores that use either a master-detail relationship or a simple schema. Stores using a master-detail relationship consist of two tables in a database. Stores using a simple schema consist of a single table in a database.

The STOREXX_SCHEMA_TYPE parameter specifies the type of schema that a store uses.

Master-detail tables

For stores that use a master-detail relationship, DSE uses two tables in a DB2 database for persisting symbols, metadata, and tick data: a ticks table and a symbols table.

Ticks table

DSE stores ticks in this table. The STOREXX_TICK_TABLE parameter determines the name of this table. The STOREXX_SCHEMA parameter determines the schema for this table. See “Store configuration parameters” on page 74 for more information about these parameters.

Restrictions:

- The ticks table must contain the symbol ID and a timestamp.
  - The symbol ID is an integer column that is populated by DSE and is the foreign key that joins the tick records to the symbol to which the tick corresponds.
  - The ticks table is used for persistence of all fields in the tick data and can also contain additional columns, as required, that are not used by DSE. These additional columns must allow NULL values or have default values.

The ticks table must contain the symbol ID, which is an integer column that is populated by DSE. The symbol ID is the foreign key that joins the tick records to the symbol to which the tick corresponds. The ticks table is used for persistence of all fields in the tick data. Therefore, this table must include a timestamp column. This table can also contain additional columns, as required, that are not used by DSE. These additional columns must allow NULL values or have default values.

See Chapter 10, “Supported data types,” on page 83 for more information on the supported data types and the compatibility of the SQL, C and DSE data types.

Symbols table

DSE stores symbols and persisted metadata in this table. The STOREXX_SYMBOL_TABLE parameter determines the name of this table. The STOREXX_SCHEMA parameter determines the schema for this table. See “Store configuration parameters” on page 74 for more information about these parameters.

The symbols table must contain at least 2 columns for the symbol name and ID.
The symbol name is a varchar(32) column and is populated by DSE.
The symbol ID is a generated identity column that is populated by the
database and retrieved by DSE.

The symbols table is also used for persistence of any metadata fields that
are marked as STORE READWRITE or STORE READWRITE AUTO. The
database table column names can be different from the DSE field names if
the record metadata schema definition specifies the storage system names.
The symbols table can also contain additional columns, as required, that
are not used by DSE.

See Chapter 10, “Supported data types,” on page 83 for more information
on the supported data types and the compatibility of the SQL, C and DSE
data types.

The following example shows the relevant DSE configuration elements and the
 corresponding DB2 database table schema that is required to support them.

DSE configuration:

<table>
<thead>
<tr>
<th>STORE01_SCHEMA</th>
<th>&quot;dse&quot;</th>
<th>#Name of the schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE01_SCHEMA_TYPE</td>
<td>0</td>
<td># 0 for master-detail schema, 1 for simple schema</td>
</tr>
<tr>
<td>STORE01_SYMBOL_TABLE</td>
<td>&quot;trade_symbols&quot;</td>
<td>#Name of the symbols table</td>
</tr>
<tr>
<td>STORE01_TICK_TABLE</td>
<td>&quot;trade_ticks&quot;</td>
<td>#Name of the ticks table</td>
</tr>
<tr>
<td>STORE01_RECTYPE</td>
<td>1</td>
<td>#The record type of ticks in the Store</td>
</tr>
</tbody>
</table>

| RECORD01_NAME | "dse.trade" | |
| RECORD01_SCHEMA | "(timestamp timestamp, 
| | trade_price double, 
| | trade_volume int, 
| | sequence_no int)" |
| RECORD01_META_SCHEMA | "(last_trade_timestamp timestamp, 
| | last_trade_price double store read last_price, 
| | last_trade_volume int store read last_volume, 
| | high_price double store read with default -1, 
| | low_price double store read with default -1, 
| | cum_volume bigint store read with default 0, 
| | num_trades int store read with default 0, 
| | last_sequence_no int not null with default -1)" |

DB2 database table schema:

```sql
CREATE TABLE dse.trade_symbols ( 
  SYMBOL_ID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY, 
  SYMBOL_NAME VARCHAR(32) NOT NULL, 
  PRIMARY KEY(SYMBOL_ID), 
  UNIQUE(SYMBOL_NAME), 
  last_price DOUBLE, 
  last_volume INTEGER, 
  high_price DOUBLE, 
  low_price DOUBLE, 
  cum_volume BIGINT, 
  num_trades INTEGER 
);
```

```sql
CREATE TABLE dse.trade_ticks ( 
  SYMBOL_ID INTEGER NOT NULL, 
  timestamp TIMESTAMP NOT NULL, 
  trade_price DOUBLE, 
  trade_volume INTEGER, 
  sequence_no INTEGER NOT NULL 
);
```
Simple schema tables

For stores that use a simple schema, DSE uses a single ticks table for persisting tick data.

DSE uses the STOREXX_SCHEMA and STOREXX_TICK_TABLE parameters to specify the name of the ticks table.

**Restriction:** The ticks table is used for persistence of all fields in the tick data and can also contain additional columns, as required, that are not used by DSE. These additional columns must allow NULL values or have default values.

The supported data types are the same as for master-detail tables.
Chapter 18. Partitioning with the DB2 Data Partitioning Feature (DPF)

The DB2 database system supports partitioned instances, using a feature called the Database Partitioning Feature (DPF). Using DPF with DSE enables significant performance improvements.

A database partition is part of a database that consists of its own data, indexes, configuration files, and transaction logs. A partitioned database is a database with two or more partitions. Tables can be located in one or more database partitions. Processors associated with each database partition are used to satisfy table requests. Data retrieval and update requests are decomposed automatically into sub-requests, and they are executed in parallel among the applicable database partitions. The DB2 database system allows a lot of flexibility in spreading data across multiple database partitions (nodes) of a partitioned database.

Users can choose how to distribute their data by declaring distribution keys. They can determine which and how many database partitions their table data can be spread across by selecting the database partition group and table space in which the data should be stored.

DSE supports two modes of partitioning of the DB2 database ticks table:

**Per-symbol partitioning**
- Ticks are partitioned based on the symbol of each tick. The hash distribution key is on the symbol ID field.

**Per-tick partitioning**
- Ticks are partitioned based on any field. Partitioning can also be based on multiple fields. For example, ticks can be partitioned based on the symbol ID and timestamp fields.

**Requirement:** Any field that is used as a partitioning key must be part of a primary key on the ticks table.


DSE flushers and DB2 database partitions

The number of DSE flushers that you configure for a store is based on the number of partitions in the DB2 database.

When using DB2 storage system partitioning with DPF, tick data is assigned to flusher threads based on the DB2 partitioning distribution hash map. Each DSE flusher thread is allocated a database partition for the ticks table to which it is flushing. The flusher thread attaches directly to its assigned node and inserts tick data only for that node.

**Requirement:** The number of flushers that are configured for the store must be greater than the number of partitions that are used by the database ticks table.
An integer multiple is recommended. For example, if your ticks table is distributed across 4 partitions, configure at least 4 flushers. Values that are multiples of 4 (such as 4, 8, 12, 16, or higher) are recommended.

**Configuring DSE for partitioning in an MPP environment**

To efficiently use a multi-partitioned database that uses the DB2 massively parallel processing (MPP) database architecture, set the value of the STORESYSXX_INFO parameter to MPP=1. This setting enables DSE to connect to each partitioning server directly to provide better performance when it flushes ticks to the persistent store.

Set up the DB2 storage system to enable this direct connection. For each partition server that the store uses, a DB2 node must be catalogued for it. The database must be catalogued at that node with the following database alias naming convention:

{STORESYSXX_CONN}{partition number}

**Example:** If the STORESYS01_CONN configuration parameter has the value RTLTEST, and the tick table for STORE01 uses the partition numbers 0, 2, and 4, then the database aliases must be:

- **RTLTEST0**
  - For node 0.
- **RTLTEST2**
  - For node 2.
- **RTLTEST4**
  - For node 4.

**Configuring WebSphere MQ with DSE in restartable mode**

If DSE is running in restartable mode, with WebSphere MQ as the transaction manager, you need to add XAResourceManager stanzas to the qm.ini configuration file for WebSphere MQ. One XAResourceManager stanza is required for each database partition.

The following table describes what is required in the qm.ini file for different types of DB2 databases, and where to find sample qm.ini files for each type:

<table>
<thead>
<tr>
<th>Table 13. XAResourceManager stanzas in the qm.ini file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of DB2 database</strong></td>
</tr>
<tr>
<td>Single partitioned database</td>
</tr>
<tr>
<td>Multi-partitioned database on one host system</td>
</tr>
<tr>
<td>Multi-partitioned database on multiple host systems</td>
</tr>
</tbody>
</table>

For more information about running DSE in restartable mode, see “Restartable mode for high availability,” on page 87.
Chapter 19. Enabling error recovery for storage systems

Errors that occur in storage systems can cause DSE failures. These failures can cause DSE to mark the store as offline and to stop trying to access the store until it is back online. You can set parameters to configure how DSE handles storage system errors.

A storage system can experience an error where the connection between the storage system and DSE remains operational, but the attempted storage system operation failed. A storage system can also experience an error that breaks the connection between the storage system and DSE, and the connection must be reestablished.

You can configure DSE to retry failed storage system operations, or attempt to reconnect to the storage system and then retry the failed storage system operation. Doing so enables DSE to attempt to recover from failures automatically for a defined number of times before it marks the store as offline.

If all attempts to retry failed operations or reconnect to storage systems fail, and the storage system error persists, then DSE disconnects from the storage system and marks the store as offline. After you correct the storage system error, you can bring the store back online by issuing the rtlmode -u command. DSE attempts to reconnect to the storage system and continue processing operations for the store.

Restriction: DSE cannot retry storage system operations or reconnect to data stores during startup.

To enable error recovery for storage systems, set the following storage system configuration parameters to define how DSE retries failed operations and reconnects after losing a connection to the storage system:

- STORESYSXX_RECONNECT_COUNT
- STORESYSXX_RECONNECT_PERIOD
- STORESYSXX_RETRY_COUNT
- STORESYSXX_RETRY_PERIOD

For information about the storage system configuration parameters, see "Storage system configuration parameters” on page 71.

For information about enabling high availability and error recovery for high availability storage systems (such as DB2 databases), see “Enabling high availability storage systems for DSE” on page 99.
Chapter 20. Correcting and reloading bad data

Data feeds can contain data that is badly formed or that does not comply with data constraints. When DSE detects this bad data while flushing to the target store, DSE discards the bad data into a discard table. You can then analyze and correct the data, and then reload it into the target ticks table.

When DSE detects bad data, a message is logged in the rt1.log file. Complete the following tasks to configure DSE to record bad data and to correct and reload the data into the target ticks table.

1. Set the value of the STOREXX_DISCARD_ENABLE parameter to 1. See “Store configuration parameters” on page 74 for more information about these parameters.
2. Create a discard table to hold the bad data from DSE shared memory. See “Creating a discard table for bad data.”
3. Ensure that the storage system interface is defined. (The DB2 storage system implementation that DSE provides defines this interface.) DSE discards the bad data into the discard table using the rtl_store_discard_tick_set() function. See “rtl_store_discard_tick_set()” on page 225 for more information.
4. Use the rtldiscard utility to convert the contents of the discard table into a delimited ASCII (DEL) file. You can now analyze and correct the ticks data. See “rtldiscard utility” on page 31 for more information.
5. Use an import utility (such as the DB2 import utility) to load the ticks data from the DEL file into the target ticks table.

You can view statistics for data that is discarded from shared memory using the rtlstat -e utility option. See “rtlstat utility” on page 43 for more information.

Creating a discard table for bad data

Create a discard table to hold bad data from DSE shared memory. This table is required if the STOREXX_DISCARD_ENABLE parameter is set to 1.

The following restrictions apply to discard tables:

- Each store can have one discard table.
- You must create the target ticks table and the discard table in the same database.
- The name of the discard table is schema.tick_table_discard, where schema is the value of the STOREXX_SCHEMA parameter, and tick_table is the value of the STOREXX_TICK_TABLE parameter.

For example, if STORE01_SCHEMA='dse' and STORE01_TICK_TABLE='trade_ticks' then the name of the discard table is dse.trade_ticks_discard.

- The discard table has the following columns:
  - A tick_id column.
  - A symbol_id column
  - Columns representing each field of the tick schema. These columns must be of type VARCHAR FOR BIT DATA. The name of each of these columns must be identical to the name of the ticks column in the ticks table.

Each column in the discard table stores the corresponding tick field value that was discarded from DSE shared memory.
DSE provides an SQL file called `sample-discard-table.sql` (in the examples/simple/setup subdirectory) that you can use to create the discard table. This sample file can be used with the sample ticks table that is created by the `db2-schema.sql` file located in the same subdirectory. Modify the `sample-discard-table.sql` file to create your own discard table.

The following table lists the DSE data types and the corresponding DB2 SQL data types.

<table>
<thead>
<tr>
<th>DSE data type</th>
<th>DB2 SQL type for the discard table</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>VARCHAR(4) FOR BIT DATA</td>
</tr>
<tr>
<td>smallint</td>
<td>VARCHAR(2) FOR BIT DATA</td>
</tr>
<tr>
<td>bigint</td>
<td>VARCHAR(8) FOR BIT DATA</td>
</tr>
<tr>
<td>real</td>
<td>VARCHAR(4) FOR BIT DATA</td>
</tr>
<tr>
<td>smallfloat</td>
<td>VARCHAR(4) FOR BIT DATA</td>
</tr>
<tr>
<td>double</td>
<td>VARCHAR(8) FOR BIT DATA</td>
</tr>
<tr>
<td>float</td>
<td>VARCHAR(8) FOR BIT DATA</td>
</tr>
<tr>
<td>double(1-24)</td>
<td>VARCHAR(4) FOR BIT DATA</td>
</tr>
<tr>
<td>double(25-52)</td>
<td>VARCHAR(8) FOR BIT DATA</td>
</tr>
<tr>
<td>time_t</td>
<td>VARCHAR(16) FOR BIT DATA</td>
</tr>
<tr>
<td>timestamp(time_t)</td>
<td>VARCHAR(16) FOR BIT DATA</td>
</tr>
<tr>
<td>date</td>
<td>VARCHAR(8) FOR BIT DATA</td>
</tr>
<tr>
<td>time</td>
<td>VARCHAR(8) FOR BIT DATA</td>
</tr>
<tr>
<td>timestamp</td>
<td>VARCHAR(16) FOR BIT DATA</td>
</tr>
<tr>
<td>timestamp(rtl_datetime)</td>
<td>VARCHAR(16) FOR BIT DATA</td>
</tr>
<tr>
<td>timestamp(rtl_java_date)</td>
<td>VARCHAR(16) FOR BIT DATA</td>
</tr>
<tr>
<td>boolean</td>
<td>VARCHAR(1) FOR BIT DATA</td>
</tr>
<tr>
<td>char(n)</td>
<td>VARCHAR(n) FOR BIT DATA</td>
</tr>
<tr>
<td>varchar(n)</td>
<td>VARCHAR(n) FOR BIT DATA</td>
</tr>
<tr>
<td>byte(n)</td>
<td>VARCHAR(n) FOR BIT DATA</td>
</tr>
</tbody>
</table>

The following sample is an excerpt from the `sample-discard-table.sql` file:

```sql
CREATE TABLE rtl.ticks_discard (  
    TICK_ID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY,  
    SYMBOL_ID INTEGER NOT NULL,  
    TSTAMP VARCHAR(16) FOR BIT DATA,  
    TRDPRC_1 VARCHAR(8) FOR BIT DATA,  
    BID VARCHAR(8) FOR BIT DATA,  
    ASK VARCHAR(8) FOR BIT DATA,  
    IRGPRC VARCHAR(8) FOR BIT DATA,  
    INSPRC VARCHAR(8) FOR BIT DATA,  
    TRDVOL_1 VARCHAR(4) FOR BIT DATA,  
    ACVOL_1 VARCHAR(4) FOR BIT DATA,  
    NUM_MOVES VARCHAR(4) FOR BIT DATA,  
    BIDSIZE VARCHAR(4) FOR BIT DATA,  
    ASKSIZE VARCHAR(4) FOR BIT DATA,  
    IRGVOL VARCHAR(4) FOR BIT DATA,  
    INSVOL VARCHAR(4) FOR BIT DATA,  
    CleanseThisField VARCHAR(4) FOR BIT DATA,  
) 
```
\begin{verbatim}
ACT_FLAG1 VARCHAR(2) FOR BIT DATA,
IRGCOND VARCHAR(4) FOR BIT DATA,
INSCOND VARCHAR(4) FOR BIT DATA
\end{verbatim}
Chapter 21. SQL queries with the DSE wrapper

You can query the data in DSE shared memory using SQL through the DSE wrapper. The wrapper enables you to create a nickname in the DB2 database system, which you can query using SQL to retrieve data from DSE. The DB2 database that is running the DSE wrapper can be on a remote system or on the same system as the DSE instance that is being queried.

Prerequisite: Before you can use the wrapper, you must install it using the DSE installation program. See Chapter 4, “Installing DSE,” on page 11 for more information.

When you run a query against a nickname for a store, DSE checks shared memory for the data first. If the data is not in shared memory and DSE determines that the data is on disk, it checks for the data on disk using the appropriate storage system. DSE automatically determines if it needs to retrieve the data from shared memory, the storage system, or both, to satisfy the query.

Optionally, the nickname can be created to search shared memory only, or the storage system only. Queries perform best when they search for data that is entirely in shared memory. For example, if shared memory holds the data since 9:00 a.m., queries that specify a date and time range at or after 9:00 a.m. run the most efficiently. If you specify a date and time range earlier than 9:00 a.m., data must be read from the storage system.

A nickname is associated with a wrapper, server, and user mapping objects. These objects must be created before the nickname is created. The examples/simple/setup subdirectory contains examples of how to create a nickname and the wrapper, server, and user mapping objects. See Chapter 22, “Wrapper objects,” on page 141 for more information about these wrapper objects and their options.

Related tasks
- Chapter 26, “Writing client applications,” on page 275
  You can write C or Java client applications that connect directly to DSE to query and update data and retrieve statistics.

Related reference
- Chapter 25, “DSE interfaces,” on page 175
  DSE defines interfaces that you must implement, for feed handlers, transports, storage systems, and high availability communications.
- Chapter 27, “DSE API reference,” on page 285
  DSE provides a number of application programming interface (API) functions.

Supported query predicates

The DSE wrapper passes the values for predicates involving the symbol and timestamp columns to the remote DSE server for processing.
The following table summarizes the operations that the DSE wrapper can handle.

<table>
<thead>
<tr>
<th>Nickname column number</th>
<th>Nickname column description</th>
<th>Operations that are supported by the DSE wrapper</th>
<th>Operations that are not supported by the DSE wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>symbol</td>
<td>=  BETWEEN IN &lt;= &lt; &gt;= &lt; &gt;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>timestamp</td>
<td>BETWEEN &lt;= &gt;= IN &lt; &gt;</td>
<td></td>
</tr>
</tbody>
</table>

The DSE wrapper handles a predicate in the WHERE clause:
- The expression references the first or second column in the nickname (the symbol or timestamp column).
- The operation is supported (as indicated in the table above).
- Multiple predicates, if they exist, are joined by the AND operation.

The DSE wrapper does not handle a predicate if the following conditions are met:
- The expression references a column other than the first or second column in the nickname (the symbol or timestamp column).
- The expression references the first or second column in the nickname, but the operation is not supported (as indicated by the table above).
- Multiple predicates, if they exist, are joined by the OR operation.

If the DSE wrapper cannot handle a predicate, the DB2 database system handles it instead. However, the DSE wrapper passes back all candidate data to the DB2 database system. Depending on the query, the amount of data that is passed from the DSE wrapper to the DB2 database system can be large and the query time can be long.

**Multiple predicates**

The DSE wrapper can handle multiple predicates that are joined by the AND operation.

For example, the wrapper directly handles all predicates in the following two queries, passing the values to DSE:

```sql
SELECT * from tick_data
WHERE tick_symbol='999'
AND tick_time between '2001-02-05-14.32.19.0005' and '2001-02-05-14.32.20.0005'
```
SELECT * from tick_data
WHERE tick_symbol='999'
AND tick_time > '2001-02-05-14.32.19.0005'

In the following query, the DSE wrapper handles the predicates on the tick_symbol and tick_time columns, while the DB2 database system processes the predicate on the bidsize column:

SELECT * from tick_data
WHERE tick_symbol='999'
AND tick_time between '2001-02-05-14.32.19.0005' and '2001-02-05-14.32.20.0005'
AND bidsize=10

The DSE wrapper does not handle predicates that are joined by the OR operation. For example, the DSE wrapper does not support the following query:

SELECT * from tick_data
WHERE tick_symbol='999'
OR tick_time between '2001-02-05-14.32.19.0005' and '2001-02-05-14.32.20.0005'

However, the OR query still succeeds. The DSE wrapper retrieves all data from the remote DSE server and returns it to the DB2 database system. The DB2 database system then processes the predicates.

### DSE wrapper error codes

The DSE wrapper returns some error codes.

These codes are described in the following table:

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
</table>
| SQL1822N   | This error code is returned in the following situations:
|            | - An internal DSE error occurred. For example, the wrapper cannot attach to the remote DSE server.
|            | - An invalid value for an option was used. For example, TRACE_LEVEL < 0.
|            | The first 5 characters of the error message contains a SQLSTATE code. |
| SQL1881N   | An option is invalid. For example, the DSE wrapper does not recognize an option named FOOEY. |
| SQL1883N   | A required option is missing. For example, the DSE server requires the storagesys_name and store_name options. |
Chapter 22. Wrapper objects

DSE provides a wrapper that enables you to query data in DSE from a DB2 database using SQL. You can create DSE wrapper objects to set up communication between a DB2 database and DSE.

The DSE wrapper objects are:

**Wrapper**
Provides the mechanism for the DB2 database system to interact with a remote data source, such as a DSE server.

**Server**
Defines a DSE data source for the wrapper.

**User mapping**
Maps a DB2 user to a remote DSE server user.

**Nickname**
Maps a local DB2 object to a DSE object, such as store tick data.

---

**Wrapper**

Creating the wrapper registers the DSE wrapper in a DB2 database so that the DB2 database can interact with DSE.

**Syntax**

```
CREATE WRAPPER wrapper_name
library 'librtl.so'
options (DB2_FENCED 'N', TRACE_FILE 'full_path', TRACE_LEVEL 'level'))
```

**Options**

`DB2_FENCED 'N'`
Required. This option is not supported in this release, so this option must be set to 'N'.

`TRACE_FILE 'full_path'`
Optional. Specifies the full path to an external file for trace messages. If this option is specified, then TRACE_LEVEL must also be specified.

`TRACE_LEVEL 'level'`
Optional. Specifies the debugging level as a positive integer. The higher the debugging level, the greater the detail. Higher debugging settings include messages from lower levels:

- `level>= 20`
  Function entry and exit messages are output to the trace file.

- `level>= 30`
  The values of wrapper class members are output to the trace file.

- `level>= 40`
  Data values that are fetched from the remote DSE server are output before results are returned to the DB2 database.

If this option is specified, then TRACE_FILE must also be specified.
Important: The wrapper debug options, TRACE_FILE and TRACE_LEVEL, can generate considerable output and cause queries to run more slowly. Use these options only for problem determination.

Sample usage

The following command creates a DSE wrapper called "MY_DSE":
create wrapper MY_DSE library 'librtl.so' options (DB2_FENCED 'N');

The following command creates a DSE wrapper with debugging enabled:
create wrapper RTL_Debug library 'librtl.so'
options (DB2_FENCED 'N', TRACE_FILE '/tmp/rtl_40.trc', TRACE_LEVEL'40');

Alter

The ALTER WRAPPER command is not supported in this release.

Server

A DB2 wrapper server identifies which remote DSE server to use.

Syntax

CREATE SERVER server_name WRAPPER wrapper_name
options (HOST_NAME 'name', SERVICE_NAME 'service')

Options

HOST_NAME 'name'
Required. Specifies the host where the DSE server is running.

SERVICE_NAME 'service'
Required. Specifies an entry in the /etc/services that indicates the port on which the remote DSE server is running. Ensure that the port for this service name matches the port for the RTL_SERVICE_NAME entry in the remote DSE server configuration file.

Sample usage

The following command creates a DB2 wrapper server that attaches to a remote DSE server that is running on the same machine:
create server TSDATA wrapper MY_DSE
options (host_name 'localhost', service_name 'db2_rtl');

The server is created for a specific DB2 wrapper. You can create a server that outputs debugging messages by creating it for a debugging wrapper:
create server TSDATA_Debug wrapper RTL_Debug
options (host_name 'localhost', service_name 'db2_rtl');

Alter

The host_name and service_name that are specified for a DB2 wrapper server can be changed with the ALTER SERVER command. For example:
alter server TSDATA
options (set host_name 'my_machine.com', set service_name 'my_dse')
**User mapping**

A user mapping tells the DSE wrapper the user ID and password to use when connecting to the remote DSE server.

**Syntax**

```sql
CREATE USER MAPPING for local_user SERVER server_name
options (REMOTE_AUTHID 'dse group name', REMOTE_PASSWORD 'password')
```

**Options**

- **REMOTE_AUTHID 'dse group name'**
  - Required. Specifies which DSE group to use. `dse group name` must match a GROUPXX_NAME entry in the remote DSE server configuration file.

- **REMOTE_PASSWORD 'password'**
  - Required. Specifies the password for the group that is provided by the REMOTE_AUTHID option. `password` must be the unencoded password that corresponds to the encoded GROUPXX_PASSWORD entry in the remote DSE server configuration file. The DB2 database encodes this value and stores it in an unreadable format in the system catalogs.

**Sample usage**

The following command provides a remote ID and password for the local user "kevin":

```sql
create user mapping for kevin server TSDATA
options (remote_authid 'rtltest', remote_password 'marshmello');
```

The user mapping is created for a specific DB2 wrapper server. We can create a user mapping that outputs debugging messages by creating it for a debugging server:

```sql
create user mapping for kevin server TSDATA_debug
options (remote_authid 'rtltest', remote_password 'marshmello');
```

**Alter**

You can modify the remote_password and remote_authid setting with the ALTER USER MAPPING command. For example, the following command switches the mapping for user "kevin" to a different DSE group and password:

```sql
alter user mapping for kevin server TSDATA
options (set remote_authid 'rtltest2', set remote_password 'AintItFun');
```

**Nickname**

A nickname maps a local DB2 schema to the remote DSE data store. The DSE wrapper retrieves data from the remote DSE server and returns it to the DB2 database in the SQL types that the DB2 database knows how to manage.

Creating a nickname has the following objectives:

- Identifies the storage system and data store to use at the remote DSE server
- Maps a local schema to the remote tick or metadata structure
- Returns software release information

Nicknames cannot be created for stores that use a simple schema.
Syntax

The CREATE NICKNAME syntax varies depending on the record type. The three record types are:

- TICKDATA
- METADATA
- RELEASE

The record type can be specified with the RECTYPE nickname option. The default record type is TICKDATA.

**TICKDATA record type**

```
CREATE NICKNAME name
(column_name varchar(32),
 column_name timestamp [OPTIONS (SRC_COL_NAME 'column name value')] 
[, column_name column_type [OPTIONS (SRC_COL_NAME 'column name value')]])
```

for SERVER server
options
{  
   STORAGESYS_NAME 'name',
   STORE_NAME 'name'
   RECTYPE 'TICKDATA'
   , RTLSCAN 'RTLSCAN_INMEM_ONLY|RTLSCAN_STORE_ONLY'
   , TRACE_LEVEL 'new level'
}

The first two columns are required in a TICKDATA nickname:

- The first column must be a VARCHAR and must be 32 bytes long. This column is used for predicates in the WHERE clause that search by symbol name. The DSE wrapper copies the symbol name for the data to this column when returning results to the DB2 database system.
- The second column must be a timestamp. It is used for predicates in the WHERE clause that search by time. The DSE wrapper copies the first element of a tick structure to this column when returning results to the DB2 database system.

The mapping between the nickname and the tick data structure is case sensitive if:

- The column name in the tick structure is enclosed in double quotation marks.
- The value of the SRC_COL_NAME option is enclosed in double quotation marks.

In all other cases, the columns are mapped without case sensitivity.

**METADATA record type**

```
CREATE NICKNAME name
(column_name varchar(32),
 [, column_name column_type [OPTIONS (SRC_COL_NAME 'column name value')]])
```

for SERVER server
options
{  
   STORAGESYS_NAME 'name',
   STORE_NAME 'name',
   RECTYPE 'METADATA'
   [, TRACE_LEVEL 'new level']
}

The first column in a metadata nickname must be a VARCHAR and must be 32 bytes long. This column is used for predicates in the WHERE clause that search by symbol name. The DSE wrapper copies the symbol name for the data to this column when returning results to the DB2 database system.

The mapping between the nickname and the metadata structure is case sensitive if:

- The column name in the metadata structure is enclosed in double quotation marks.
- The value of the SRC_COL_NAME option is enclosed in double quotation marks.

In all other cases, the columns are mapped without case sensitivity.

**RELEASE information record type**

```sql
CREATE NICKNAME name
for SERVER server
options (RECTYPE 'RELEASE')
```

Do not specify column information. The DSE wrapper automatically adds two VARCHAR columns, one named rtl_wrapper, which returns DSE wrapper release information, and the other named rtl_api, which identifies the DSE API release that the DSE wrapper was compiled with.

**Options**

**RECTYPE (optional)**

Specifies the kind of record to which the nickname is mapped. Valid settings are:

- **TICKDATA**: The nickname schema maps to tick data records at the remote DSE server. **TICKDATA** is the default.
- **METADATA**: The nickname schema maps to metadata records at the remote DSE server.
- **RELEASE**: The nickname maps to software release information.

**RTLSCAN (optional for TICKDATA)**

Specifies if data should be retrieved from just memory ('RTLSCAN_INMEM_ONLY') or just disk ('RTLSCAN_STORE_ONLY'). The default is to retrieve data from both.

**SRC_COL_NAME (optional for TICKDATA and METADATA)**

Maps a column name in the nickname to a field name in a tick data record or metadata record. Use this option if the column name in the nickname does not match the field name in the tick data record or metadata record.

**STORAGESYS_NAME (required for TICKDATA and METADATA)**

Specifies which DSE storage system to open. It must match a STORESYSXX_NAME entry in the remote DSE server configuration file.

**STORE_NAME (required for TICKDATA and METADATA)**

Specifies the store name from which to retrieve data. It must match a STOREXX_NAME entry in the remote DSE server configuration file.

**Note**: The store must use a master-detail schema and cannot be a store that uses a simple schema.

**TRACE_LEVEL (optional)**

Overrides the TRACE_LEVEL setting for the DSE wrapper. One intended use is to turn tracing off at the nickname level.
Sample usage
TICKDATA and METADATA record types

Example 1:
This simple example maps the nickname to the simple example tick structure that is distributed with DSE:

```sql
create nickname tick_data
(symbol    varchar(32),
tstamp     timestamp,
trdprc_1   double,
bid        double,
ask        double,
irgprc     double,
insprc     double,
trdvol_1   int options,
acvol_1    int options,
um_moves   int,
bidsize    int,
asksize    int,
irgvol     int,
insvol     int,
cleansethisfield int,
act_flag1  char(2),
irgcond    char(4),
inscond    char(4)
)
for SERVER TSDATA
options(storagesys_name 'db2', store_name 'rtl.ticks');
```

Example 2:
You can use the SRC_COL_NAME option to map a column in your nickname to a field in the tick data record or metadata record. In the following example, the name of the "trade price" column in the tick_data nickname is trdprc. This column name is mapped to the field name TRDPRC_1 in the tick data record rtl.ticks:

```sql
create nickname tick_data
(symbol    varchar(32),
tstamp     timestamp,
trdprc     double options (SRC_COL_NAME 'TRDPRC_1'),
ask        double,
bidsize    int,
asksize    int,
irgvol     int,
insvol     int,
cleansethisfield int,
act_flag1  char(2),
irgcond    char(4),
inscond    char(4)
)
for SERVER TSDATA
options(storagesys_name 'db2', store_name 'rtl.ticks');
```

Example 3:
In this example, the SRC_COL_NAME option maps multiple columns in the nickname to a single field in the tick data record or metadata record. The SRC_COL_NAME option maps the bid1 and bid2 columns in the tick_data nickname to the bid field in the tick data record dsel.store_1.

```sql
create nickname tick_data
(symbol    varchar(32),
tstamp     timestamp,
trdprc_1   double,
bid1       double options (SRC_COL_NAME 'bid'),
bid2       double options (SRC_COL_NAME 'bid'),
ask        double,
irgprc     double,
```
When mapping subsets of tick data fields, remember that the first column in the nickname must be a VARCHAR for the symbol name and the second column must be a timestamp for the first element in the tick data.

**Example 4:**

In this example, the mapping between some of the columns in nickname and the tick or metadata structure is case sensitive because:

- The field name in the tick or metadata structure is enclosed in double quotation marks.
- The value of the SRC_COL_NAME option is enclosed in double quotation marks.

```
CREATE NICKNAME meta_data
  (symbol VARCHAR(32),
   "Last_Trade_Price" double,
   LOW_PRICE double OPTIONS(SRC_COL_NAME "Low_Price"),
   NUM_TRADES int
  )
for SERVER MYSERVER
options (STORAGESYS_NAME 'db2', STORE_NAME 'MYSTORE', RECTYPE 'METADATA');
```

This nickname maps to the following record definition:

```
RECORD66_META_SCHEMA
 "(last_trade_timestamp timestamp,
  "Last_Trade_Price" double,
  last_trade_volume int,
  high_price double,
  "Low.Price" double,
  cum_volume bigint,
  num_trades int,
  last_sequence_no int
)"
```

In this example, the following mapping between columns occurs:

<table>
<thead>
<tr>
<th>Nickname column</th>
<th>Metadata structure field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last_Trade_Price</td>
<td>Last_Trade_Price</td>
</tr>
<tr>
<td>LOW_PRICE</td>
<td>Low_Price</td>
</tr>
<tr>
<td>NUM_TRADES</td>
<td>num_trades</td>
</tr>
</tbody>
</table>

In the following example, the mapping between the nickname column LOW_PRICE and the metadata structure field Low_Price fails because:
The metadata field name (in the RECORD66_META_SCHEMA definition) is enclosed in quotation marks.

CREATE NICKNAME meta_data
(symbol VARCHAR(50),
 LOW_PRICE  double)
for SERVER MYSERVER \ 
options (STORAGESYS_NAME 'db2', STORE_NAME 'MYSTORE', RECTYPE 'METADATA');

RELEASE record type

Create a nickname for obtaining software release information as shown below:

CREATE NICKNAME release_info
for SERVER TSDATA options(rectype 'RELEASE')

The DSE wrapper automatically adds two columns:

rtl_wrapper
Provides release information for the DSE wrapper.

rtl_api Identifies the release of the DSE API with which the DSE wrapper was compiled.

For example:

select rtl_wrapper from release_info

RTL_WRAPPER
--------------------------------------------------
DSE Wrapper 9.01 (Tue Apr 18 08:35:30 PDT 2006)
  1 record(s) selected.

select rtl_api from release_info

RTL_API
--------------------------------------------------
DSE API 9.1.0 (Tue Apr 18 08:35:30 PDT 2006)
  1 record(s) selected.

Alter

The ALTER NICKNAME command is not supported in this release.

rtlElem UDF

The DSE API provides the rtlElem function to return the row that is the closest match for the specified symbol, time, and boolean operator.

The DSE wrapper recognizes a user-defined function (UDF) named rtlElem. If it is present in a query, the DSE wrapper retrieves at most a single row instead of the normal result set.

rtlElem isn't a real UDF; you cannot execute it with the VALUES statement and cannot execute it with non-DSE nicknames or tables. But it must be declared to the DB2 database system for it to be allowed in queries.
Syntax

Create the rtlElem UDF as shown below:

```sql
create function rtlElem
  (sym_col varchar(), symbol varchar(), search_time timestamp, ops varchar())
returns integer
  as template deterministic no external action;
```

You must name the UDF rtlElem because that is the name that the DSE wrapper looks for. However, DB2 object names are case insensitive, so you can use rtlelem, RTLELEM, RtLeLeM, or any other combination of upper and lower case characters you like.

You can create the UDF in a schema as shown below:

```sql
create function rtl.rtlElem
  (sym_col varchar(), symbol varchar(), start timestamp, ops varchar())
returns integer
  as template deterministic no external action;
```

If you specify a schema when you create the UDF, you must reference the schema when you execute the UDF or put that schema in your DB2 path as shown below:

```sql
SET CURRENT FUNCTION PATH rtl, SYSTEM PATH
```

Arguments and return value

The UDF takes four arguments:

- **symbol column**
  This is a required argument and must reference the symbol column in the nickname.

- **symbol**
  This is a required argument and is a character string with the symbol for which to search.

- **search time**
  This is a required argument and is the timestamp for which to search.

- **boolean search operator**
  This is a required argument and is a text value containing <, <=, =, >, or >=.

The UDF integer return value is not currently used by the DSE wrapper but is required for the query invocation.

Sample usage

Execute the UDF like this:

```sql
select cast(symbol as char(4)) as symbol,
  tick_time,
  decimal(trdprc, 6, 2) as trdprc,
  decimal(bid, 6, 2) as bid,
  decimal(ask, 6, 2) as ask,
  decimal(irgprc, 6, 2) as irgprc,
  decimal(insprc, 6, 2) as insprc
from tick_data
where rtlElem(symbol, '2194', TIMESTAMPTZ('2001-02-05-14.32.19.0005'), '<') = 1
```

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>TICK_TIME</th>
<th>TRDPRC</th>
<th>BID</th>
<th>ASK</th>
<th>IRGPRC</th>
<th>INSPRC</th>
</tr>
</thead>
</table>
1 record(s) selected.
Part 5. Developing DSE components and applications

DSE provides interfaces and APIs to enable you to develop DSE components, such as feed handlers and transports, and application programs.
Chapter 23. Feed handlers

DSE uses feed handlers to receive and process data from different kinds of data feeds. The feed handler defines how the data is formatted and processed for the data feed.

DSE can process many different types of data feeds, such as financial market data, RFID, and sensor data. The feed handler must understand the feed data format and the processing that is required. The transport handles the feed communication transport.

The transport can handle many different feed communication transports, such as files, sockets and Websphere MQ queues. The feed handler extracts ticks from the messages received by the transport and processes them. DSE works with the feed handler to store ticks, symbols, and symbols metadata in shared memory and to flush this data to the DB2 storage system.

Figure 3 shows the relationship between transports and feed handlers.

The feed handler performs the following functions:

- Interprets and transforms the data format of the feed messages. See “Approaches to handling data formats” on page 154.
- Processes the data, including:
  - Storing records
  - Updating metadata
  - Publishing messages

See “Feed data processing” on page 155.
See “Feed handler interface” on page 175 for more information about the feed handler interface. An IBM consultant can help you develop a customized feed handler that interacts with data to fit your feed requirements.

See Chapter 24, “Transports,” on page 163 for more information about transport handlers, if you want to develop your own or gain a better understanding of how they work with DSE.

**Related tasks**

- Chapter 26, “Writing client applications,” on page 275
  You can write C or Java client applications that connect directly to DSE to query and update data and retrieve statistics.

**Related reference**

- Chapter 28, “Public header files,” on page 435
  DSE provides header files for developers of DSE components and applications.

---

**Approaches to handling data formats**

The feed handler must understand the messages that are received from the feed (through the transport layer) and know how to transform them to the format in which DSE handles them.

The feed handler can ignore messages or fields within messages that it does not want to process. The feed handler can also enrich or calculate fields that are not provided by the feed but that DSE will store, maintain as metadata, or publish.

The feed handler has two basic approaches for handling the data format of records that DSE handles:

**Simple approach**

- Use simple fixed structures that are defined in a C header file at compile time.
  - This approach is easy to implement, but the feed handler requires code changes and a rebuild when the data format changes.
  - The simple example (in the examples/simple subdirectory) demonstrates this approach.

**Generic approach**

- Use the DSE record information to generically handle data formats at run time.
  - This approach is harder to implement than the simple approach, but it does not require code changes and a rebuild for most data format changes.
  - The generic example (in the examples/generic subdirectory) demonstrates this approach.

---

**DSE record structures**

The feed handler handles tick and metadata records. The RECORDXX_SCHEMA and RECORDXX_META_SCHEMA parameter definitions in the configuration file defines the structure of these records.

DSE provides the rtl_get_tick_info and rtl_get_meta_info functions to get information on DSE record structures. These functions return the following information:

- The size of the record in bytes
• The number of fields
• The field information, such as type, size, and offset in the record

This information enables feed handler developers to write generic code to determine the data format at run time and handle the data format dynamically.

Requirement: For stores using a master-detail relationship, tick records must have a timestamp as the first field. It is recommended that you use Coordinated Universal Time (UTC) instead of local time to avoid time zone and daylight savings time issues.

Feed data processing

The feed handler processes messages from the data feed and can store tick records, update metadata, and publish messages.

Tick storage

The feed handler can store a tick that was processed from a message by returning a symbol, tick record, and store ID for the tick from the rtl_feed_get_tick_from_message function.

Note: The symbol is not required if the tick store uses a simple schema.

The feed handler returns the appropriate information from these functions, and DSE stores the tick. A feed can store ticks in many different stores, but each tick can be stored in only one store.

Metadata updates

The feed handler can update metadata by returning a symbol and a store ID for the metadata from the rtl_feed_get_tick_from_message function.

DSE then invokes the rtl_feed_update_metadata function, passing the metadata record to be updated to this function. Ensure that the implementation of the rtl_feed_update_metadata function performs the following actions:
• Updates the metadata record in place
• Notifies DSE if it modifies any fields that are marked as persisted with STORE WRITE or STORE WRITE AUTO in the metadata record schema definition in the configuration file

The field information for the metadata record includes an indication of whether or not the field is persisted. Use the flags parameter of the rtl_feed_update_metadata function to notify DSE of changes.

Restriction: The metadata store must use a master detail schema. Metadata is not supported for a simple schema store.

Message publishing

The feed can publish a message by returning a non-zero message length and a message buffer from the rtl_feed_get_publish_message function.

The message to be published can be any buffer, and can be based on any of the following data:
• The message that is received by the feed
• The stored tick
• The updated metadata

Feed handler reference and context information

DSE provides API functions that enable feed handlers to retrieve information about the DSE configuration.

The HANDLERXX_INFO and FEEDXX_INFO configuration parameters allow any feed handler specific information to be configured at run time. These parameters are plain text fields that are not interpreted by DSE and can be used by feed handler developers for their own requirements.

The feed handler framework provides the use of user data areas of various scopes to allow feed handler developers to maintain their own reference and context information at the handler, feed, source and target levels. The feed handler allocates and frees these areas. API functions can be used to access these areas when they are not passed as parameters to the feed handler interface directly.

Related reference
"Feed handler API" on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.

Control flows for feed handlers

All feed handler processing follows a set of control flows. Understand the flows so that you can develop your own feed handler. The flows use the feed handler interface.

The feed and transport work closely together to process a feed. See "Transport control flows" on page 164 for additional information on the transport.

In the following feed handler processing flow, DSE is started with the feed handler defined in the configuration file. The flow applies to all feed handlers:
1. DSE initializes the feed handler. See "Feed handler initialization" on page 157.
2. DSE creates feed threads for each feed that is defined in the configuration file, using the following flow:
   a. DSE initializes the feed. See "Feed initialization" on page 157.
   b. Each feed thread repeats the following processing continually, until the feed is shut down or until there are no more messages for the feed:
      1) If DSE or the feed receives a request to shut down, DSE shuts down the feed. See "Feed shutdown" on page 158.
      2) DSE processes the next feed message. See "Per message processing" on page 157.
         If the message is processed successfully, and the feed has more messages to process, DSE continues processing feed messages.
      3) The feed handles source and target switches. It also handles requests to pause or resume the feed. See "Data source and target switching" on page 166 and "Feed pause and resume" on page 167 for more information.
   c. DSE shuts down the feed. See "Feed shutdown" on page 158.
3. DSE shuts down the feed handler. See "Feed handler shutdown" on page 158.

Related reference
The feed handler defines the interface for the data feed, handling the feed data format and the processing that is required.

**Feed handler initialization**

Feed handler initialization occurs when DSE starts, before any other processing in the feed handler is invoked. Feed handlers are initialized once for each invocation of DSE.

In the following feed handler initialization flow, DSE is started with the feed handler defined in the configuration file. The flow applies to all feed handlers:

1. DSE loads the feed handler shared library.
2. DSE invokes the rtl_handler_init function and stores the returned handler user data area for subsequent access by the feed handler.

**Feed initialization**

Feed initialization occurs when DSE starts, before any other processing in the feed is invoked. Each feed is initialized only once for each invocation of DSE.

In the following feed initialization flow, DSE creates feed threads for each feed that is defined in the configuration file. The flow applies to each feed:

1. DSE invokes the rtl_feed_init function and stores the returned feed user data area for subsequent access by the feed.
2. The feed’s transport is initialized. See "Transport initialization" on page 165 for more information.

**Per message processing**

Each feed thread in DSE continuously reads messages from the feed source and processes them. The feed thread can process many messages during each invocation of DSE.

The flow for processing messages applies to each feed:

1. DSE receives data from the feed data source. Then, DSE invokes the rtl_feed_get_tick_from_message function to get the next tick to be processed from the feed source.
2. If the tick store ID that is returned in step 1 is not -1, DSE writes the tick that is returned in step 1 to shared memory. If the tick store uses a master-detail schema, the tick is associated with the corresponding symbol in DSE shared memory.
3. If the metadata store ID that is returned in step 1 is not -1, and the store uses a master-detail schema, DSE retrieves the metadata for the symbol that is returned in step 1. If necessary, DSE creates the metadata. Then DSE invokes the rtl_feed_update_metadata function to pass the symbol metadata to be updated.
4. DSE invokes the rtl_feed_get_publish_message function.
   If the message length that is returned by the rtl_feed_get_publish_message function is greater than 0, DSE invokes the rtl_target_put_message function, passing the message buffer and length that is returned.
5. If the return code from rtl_feed_get_tick_from_message in step 1 was RTL_MORE_TICKS, repeat steps 1 through 4, incrementing the iteration count in each rtl_feed_get_tick_from_message function call, until there are no more ticks in the message.
Related reference
“Data source and target switching” on page 166
Each feed thread periodically checks for requests to switch its data source or target. Source and target switches are independent and can occur many times during each invocation of DSE.
“Feed pause and resume” on page 167
Feeds can be paused and resumed at any time. When a feed is paused, the reading of messages from a data source is temporarily suspended.

Feed shutdown
Feed shutdown occurs when DSE shuts down. No other processing in the feed is invoked after shutdown. Shutdown is carried out only once for each feed for each invocation of DSE.

The following flow for shutdown applies to each feed:
1. The feed transport is shut down. See “Transport shutdown” on page 169 for more information.
2. DSE invokes the rtl_feed_shutdown function.
   a. When this function is called, DSE passes the feed user data that is allocated by the rtl_feed_init function. The rtl_feed_shutdown function frees the feed user data.
   b. After the function returns, DSE nulls the link to the user data so that the user data can no longer be accessed by the feed.
3. DSE terminates the feed thread.

Feed handler shutdown
After DSE and the feed handler shut down, no other processing in the feed handler is invoked. Shutdown is carried out only once for each invocation of DSE.

In the following shutdown flow, DSE starts to shutdown. The flow applies to each feed handler that is defined in the DSE configuration file:
1. DSE invokes the rtl_handler_shutdown function.
   a. When this function is called, DSE passes the handler user data that is allocated by the rtl_handler_init function. The rtl_handler_shutdown function frees the handler user data.
   b. After the function returns, DSE nulls the link to the user data so that the user data can no longer be accessed by the handler.
2. DSE unloads the handler shared library.

Error handling and logging for feed handlers
When you develop a feed handler, ensure that it checks for errors and reports them. All DSE feed handler interface functions allow a status indicator to be returned to DSE. A feed handler can experience errors that it can handle itself, without DSE needing to be notified of the error. Error handling allows DSE to continue its normal operation.

DSE provides a mechanism for feed handlers to log messages into the DSE message log, where the DSE system messages are logged. This log provides a single location for all DSE diagnostic and informational messages and should be used for all feed handler logging. This mechanism consists of two functions
Use the rtl_handler_log function during feed handler initialization and shutdown in the implementations of the feed handler interface functions rtl_handler_init and rtl_handler_shutdown. Use the rtl_feed_log function in the implementations of all other feed handler interface functions. Each of these functions takes a message ID number and a variable list of arguments to be expanded into placeholders in the message to be logged.

The message file must contain the message number, the message severity, and the message text. Each message component must be separated by a colon.

- The message number is an integer between 1 and 999.
- The message severity is indicated by a single character:
  
  **E** Error
  **F** Fatal
  **I** Information
  **W** Warning

A fatal error causes DSE to exit immediately without flushing data in memory to disk.

- The message text is a format string based on the C printf function format string. The message text can contain ordinary characters that are written unchanged to the log file and conversion specifications that are replaced by arguments that are passed to the log functions.

**Example:**

See the following message:
2:E:Can't connect to "%s:%d". Errno %d

This message is logged by the following call to the rtl_feed_log function:
rtl_feed_log(feed_id, 2, l_ip_address, l_port_no, errno);

Where:

- The first parameter (feed_id) is the feed number that is passed into the feed handler interface function by DSE.
- The second parameter (2) is the message number to be logged.
- The remaining parameters are the variable arguments list to replace the printf style format string conversion specifications in the message text.
  - The l_ip_address parameter is a character string variable.
  - The l_port_no and errno parameters are integer variables.

The resulting log file output looks like this:
[2006-06-12T13:50:13-08:00] DSEF0002E: Can't connect to "127.0.0.1:4650". Errno 111

**Trace mechanism for feed handlers**

DSE provides a trace mechanism for feed handlers to add trace messages into the DSE trace output, where the DSE system trace messages are logged. This trace output provides a single location for all DSE trace messages and should be used for all feed handler tracing.
The trace mechanism enables messages to be given a level and for trace to be enabled at varying levels dynamically.

For more details on using trace at run time, see the following information:
- "Trace configuration parameters" on page 79
- The `rtlmode -D` and `rtlmode -T` commands in "rtlmode utility" on page 32

The trace mechanism consists of several C macros that DSE provides. Feed handler developers can call these macros. The macro definitions are defined in the public header file `rt1_trace.h`. Ensure that all feed handler source files that will include trace messages do the following:
- Include the trace header file.
- Define the trace component to be a feed handler, as follows:
  ```
  #define COMPONENT COMPONENT_ID_HANDLER
  ```

DSE defines the following trace macros:

**TRACE(lvl, args)**

  Writes a character string trace message of the given level for the given arguments.

**TRACEB(lvl, ptr, len)**

  Writes a binary trace message of the given level for the given binary buffer.

**TRACESUB(lvl, args)**

  Writes a subroutine entry character string trace message of the given level for the given arguments. This macro indents the trace message in the trace output until a RETURN, RETURN_PTR, or F_RETURN macro is called.

**RETURN(lvl, v)**

  Writes a subroutine return trace message of the given level for the given numeric return value. This macro removes one level of indentation of the trace message in the trace output.

**RETURN_PTR(lvl, v)**

  Writes a subroutine return trace message of the given level for the given pointer return value. This macro removes one level of indentation of the trace message in the trace output.

**F_RETURN(lvl)**

  Writes a function return trace message of the given level. This macro removes one level of indentation of the trace message in the trace output.

The `args` parameter for the TRACE and TRACESUB macros is a variable length arguments list that must start with a character string trace message.

The trace message is a format string based on the C `printf` function format string. The trace message can contain ordinary characters that are written unchanged to the trace output and conversion specifications that are replaced by arguments that are passed to the trace functions. The trace message mechanism is similar to the log message text mechanism that is described in "Error handling and logging for feed handlers" on page 158.

**Related concepts**

- Chapter 31, "DSE traces," on page 485

DSE provides a trace mechanism to aid in troubleshooting problems. Set up the trace file and settings using the trace configuration parameters.
Building and deployment of feed handlers

After you develop the feed handler, build and deploy it in your runtime environment.

DSE feed handlers are shared libraries that are loaded at run time based on the configuration file. DSE includes a make include file with platform-specific make definitions and directives that are used for building a feed handler. The examples provided by DSE include a makefile in the /handler subdirectory that can be used as a base for building your own feed handler.

To deploy your feed handler, you must configure DSE with the location of your feed handler shared library file and your feed handler message file for log messages. These file locations are defined by the HANDLERXX_LIB_PATH and HANDLERXX_MSG_PATH configuration parameters. See “Feed handler configuration parameters” on page 56 for more information.

Examples of feed handlers

DSE provides a number of feed handler examples, each of which include the feed handler source code in the /handler subdirectory.

The examples are in the following subdirectories:

DSE_INSTALL_DIR/examples/simple
A feed handler based on a simple fixed record structure.

DSE_INSTALL_DIR/examples/ha
A feed handler based on a simple fixed record structure. This example includes processing for metadata synchronization in a high availability configuration.

DSE_INSTALL_DIR/examples/generic
A generic feed handler that determines the record structure at run time based on the configuration.

DSE_INSTALL_DIR/examples/taq
A feed handler that can process NYSE Trade and Quote (TAQ) historic trade data files.

The DSE_INSTALL_DIR directory is the directory in which DSE is installed.
Chapter 24. Transports

The transport handles the feed communication transport. It simplifies the development effort required to create a feed handler by encapsulating the details of the feed transport mechanism.

The feed handler developer is responsible for understanding the feed data format and the processing required, not the communication transport that is used to deliver the feed data. The transport can handle many different feed communication transports, such as files, sockets and Websphere MQ queues. Transports handle opening, getting messages from, and closing the feed source. For feed message publishing, transports perform the similar tasks of opening, writing messages to, and closing a data target. These operations are achieved by means of the transport interface. See “Transport interface” on page 252 for more information about this interface.

Figure 4 shows the relationship between transports and feed handlers.

The information in these topics help you to implement your own transport interface. You are not required to implement your own transport interface; DSE provides three pre-built libraries for use with files, TCP/IP sockets, and WebSphere MQ queues. You can use these pre-built transports, and then develop a feed handler to process the messages that the transports obtain from the feed source.


Related tasks
You can write C or Java client applications that connect directly to DSE to query and update data and retrieve statistics.

**Related reference**

Chapter 28, “Public header files,” on page 435
DSE provides header files for developers of DSE components and applications.

### Transport properties

DSE determines the properties and capabilities of transports by invoking the rtl_transport_get_property() function. DSE obtains these properties to determine how to interact with the feed source.

The types of transport properties are:

- Delivery model
- Source switch type
- Restartability
- Message ID length

See [“rtl_transport_get_property()” on page 256](#) for more information about these properties.

### Transport control flows

All transport processing follows a set of control flows. Understand the flows so that you can develop your own transport handler or gain further perspective on how DSE uses transports in conjunction with feed handlers.

These flows use the transport handler interface. For more information about this interface, see [“Transport interface” on page 252](#). The control flows are summarized here:

1. DSE initializes the transport handler. See [“Transport handler initialization” on page 165](#).
2. DSE creates feed threads for each feed that is defined in the configuration file, using the following flow:
   a. DSE initializes the feed and its associated transport. See [“Transport initialization” on page 165](#).
   b. Each feed thread repeats the following processing continually, until there are no more messages for the feed or the feed is shut down:
      1) If the feed receives a request to switch the data source or target, DSE performs the switch at the earliest switch point. See [“Data source and target switching” on page 166](#).
      2) If the feed receives a request to pause, DSE causes the feed to stop processing until it receives a request to resume. See [“Feed pause and resume” on page 167](#).
      3) If DSE receives a request to shut down, DSE shuts down the feed and its associated transport. See [“Transport shutdown” on page 169](#).
      4) DSE obtains the next message from the data source through the transport, and passes it to the feed handler for further processing. See [“Per message processing” on page 168](#).
   c. When there are no more messages to process, DSE closes the data source and waits for a new source to be made available.
d. DSE shuts down the feed and its transport. See “Transport shutdown” on page 169.

3. If DSE is running in restartable mode, flusher threads utilize the transport to ensure transactional integrity, so that ticks are only removed from the data source (or marked as read) after they are committed to the store. DSE creates one or more flusher threads for each store that is defined in the configuration file. The following flow summarizes how transport functions are used in each flusher thread when DSE is running in restartable mode:
   a. Flushers are assigned to feeds at runtime. After this assignment is made, the flusher initializes its transport. See “Transport initialization.”
   b. Each flusher continually processes ticks in batches, until there are no more ticks to be flushed or the flusher is shut down. See “Per message processing” on page 168.
   c. DSE shuts down the flusher and its transport. See “Transport shutdown” on page 169.

4. When all feeds and flushers are shut down, DSE shuts down the transport handler. See “Transport handler shutdown” on page 169.

Transport handler initialization

Transport handler initialization occurs when DSE starts, before any other transport functions are invoked. Transport handlers are initialized once for each invocation of DSE.

In the following control flow, DSE is started with the transport handler defined in the configuration file. The flow applies to all transport handlers:
1. DSE loads the transport handler shared library.
2. DSE invokes the rtl_transport_get_property() function repeatedly to obtain information about each of these attributes of the transport:
   • What type of delivery model it supports (push or pull model)
   • What type of source switching it allows (immediate or deferred)
   • What capability it has for restartability (non-restartable or transactional)
   • How long the message IDs are (if it is a transactional transport)
3. DSE invokes the rtl_transport_handler_init() function and stores a pointer to the returned transport handler user data for subsequent access by the transport.

Transport initialization

Transport initialization occurs when DSE starts each thread that uses a transport, before any other processing in the thread is begun. Transports are initialized by each feed thread. If DSE is running in restartable mode, transports are also initialized by each flusher thread.

In the following control flow, DSE creates feed threads for each feed that is defined in the configuration file. Each feed has an associated transport, also defined in the configuration file. This flow applies to each feed thread:
1. DSE invokes the rtl_feed_get_message_fmts() function in the feed handler.
   This mechanism allows the feed handler to describe the low-level format of source and target data messages.
2. DSE invokes the rtl_transport_init() function.
   a. The message format information obtained from the feed handler in step 1 is passed to the transport.
b. A pointer to the returned transport user data is stored for subsequent access by all subsequent transport functions.

3. DSE invokes the rtl_transport_target_open() function.
   A pointer to the user data that was obtained in step 2 on page 165 is passed to the transport.

4. DSE invokes the rtl_transport_source_open() function.
   a. A pointer to the user data that was obtained in step 2 on page 165 is passed to the transport.
   b. If DSE is running in restartable mode and the transport supports transactional restartability, the source is opened in RTL_OPEN_BROWSE mode. Otherwise the source is opened in RTL_OPEN_READ mode.

5. If the transport uses a push model for message delivery, DSE invokes the rtl_transport_source_register_callback function to register itself to receive messages from the data source.

DSE creates flusher threads for each store that is defined in the configuration file. At runtime, flusher threads are assigned to a particular feed by the partitioner thread.

If DSE is running in restartable mode, the following control flow applies to each flusher thread assigned to a feed:

1. DSE invokes the rtl_transport_init() function.
   No message format information is provided to the transport (messages are only deleted by the transport from the data source). A pointer to the returned transport user data is stored for subsequent access by all subsequent transport functions.

2. DSE invokes the rtl_transport_source_open() function.
   a. The transport source is opened in RTL_OPEN_READ mode, signifying that messages are to be deleted after they are read from the source.
   b. A pointer to the user data that was obtained in step 1 is passed to the transport.

**Data source and target switching**

Each feed thread periodically checks for requests to switch its data source or target. Source and target switches are independent and can occur many times during each invocation of DSE.

The following flow switches the data source for a feed thread:

1. DSE detects that a data source switch was requested.
   - If the transport’s source switch type is RTL_DEFERRED, DSE continues processing messages from the current source until no more messages are available.
   - If the transport’s source switch type is RTL_IMMEDIATE, or no more messages are available from the current source, DSE invokes the rtl_transport_source_close() function.

2. DSE invokes the rtl_transport_source_open() function.
   The name of the new source is passed to the transport.
   If DSE is running in restartable mode and the transport supports transactional restartability, the source is opened in RTL_OPEN_BROWSE mode. Otherwise the source is opened in RTL_OPEN_READ mode.
3. If the transport uses a push model for message delivery, DSE invokes the 
rtl_transport_source_register_callback() function to register itself to receive 
messages from the data source.

If DSE is running in restartable mode, a source switch also occurs in each flusher 
thread assigned to a feed. The switch is coordinated by the feed thread to ensure 
that all messages from the old source are flushed before switching to the new 
source. The following flow switches the data source for each flusher thread 
assigned to a feed:

1. DSE invokes the rtl_transport_source_close() function.
2. DSE invokes the rtl_transport_source_open() function
   a. The name of the new source is passed to the transport.
   b. The transport source is opened in RTL_OPEN_READ mode.

The following flow switches the data target for a feed thread:

1. DSE detects that a data target switch was requested.
2. After all processing is completed for the current message, DSE invokes the 
rtl_transport_target_close() function.
3. DSE invokes the rtl_transport_target_open() function. The name of the new 
target is passed to the transport.

Only feed threads utilize the transport target functions, so there is no 
corresponding control flow for flusher threads.

Related concepts

"Per message processing" on page 157

Each feed thread in DSE continuously reads messages from the feed source and 
processes them. The feed thread can process many messages during each 
invocation of DSE.

Feed pause and resume

Feeds can be paused and resumed at any time. When a feed is paused, the reading 
of messages from a data source is temporarily suspended.

The process of pausing and resuming a feed depends on the delivery model of the 
transport:

• If the transport uses the pull model, the feed thread blocks the feed until it is 
  commanded to resume. No transport function calls are invoked when pausing or 
  resuming.

• If the transport uses the push model, DSE invokes the 
  rtl_transport_source_close() function on the feed thread when pausing. This 
  causes the data source to unregister the DSE callback function so that message 
  delivery is stopped. When the feed thread is resumed, DSE invokes the 
  rtl_transport_source_open() and rtl_transport_source_register_callback() 
  functions to reestablish the connection to the data source.

Related concepts

"Per message processing" on page 157

Each feed thread in DSE continuously reads messages from the feed source and 
processes them. The feed thread can process many messages during each 
invocation of DSE.
Per message processing

Each feed thread in DSE continuously reads messages from the data source and processes them. The feed thread can process many messages during each invocation of DSE.

The following flow for processing messages applies to each feed thread:

1. DSE obtains a message from the data source. How this is accomplished depends on the delivery model of the transport:
   - If the transport uses the pull model, DSE invokes the `rtl_transport_source_get_message()` function to get the next message to be processed from the data source.
   - If the transport uses the push model, the data source calls the DSE message callback function, which was previously registered using the `rtl_transport_source_register_callback()` function.

2. DSE passes the message to the feed handler by invoking the `rtl_feed_get_tick_from_message()` function. If the message contains multiple ticks, this function is invoked once for each tick in the message:
   - If the `rtl_feed_get_tick_from_message()` function returns `RTL_MORE_TICKS` or `RTL_LAST_TICK`, DSE generally processes the tick as follows (see “Per message processing” on page 157 for more specific information):
     a. DSE writes the tick to shared memory.
     b. DSE invokes the `rtl_feed_update_metadata()` function in the feed handler.
     c. DSE invokes the `rtl_feed_getPublish_message()` function in the feed handler.
     d. DSE invokes the `rtl_transport_targetPut_message()` function.
   - If the `rtl_feed_get_tick_from_message()` function returns `RTL_NO_TICK`, DSE invokes the `rtl_transport_source_delete_message()` function to remove the message from the data source.

3. If the `rtl_transport_source_get_message()` function returns `RTL_NO_MORE_MESSAGES`, DSE invokes the `rtl_transport_source_close()` function and idles the feed thread. When a new data source is available, DSE invokes the `rtl_transport_source_open()` function and returns to step 1.

If DSE is running in restartable mode, the following flow for processing messages applies to each flusher thread.

Ticks are flushed to a store in batches. The size of a batch is generally the value of the `STOREXX_FLUSH_BATCH` configuration parameter. However, if a data source message contains multiple ticks, all ticks from that message will be flushed in a single batch to preserve transactional integrity. For each batch of ticks to be flushed:

1. DSE invokes the `rtl_transport_source_batchBegin()` function to mark the start of a batch of ticks to be removed from the data source.
2. DSE invokes the `rtl_store_flush_batchBegin()` function to mark the start of a batch of ticks to be flushed to the store.
3. For each tick in the batch, DSE flushes the tick to the store.
   - If this is the only tick associated with a message, or is the final tick of a multiple-tick message, DSE invokes the `rtl_transport_source_delete_message()` function.
   - The message ID associated with the tick is passed to the transport so it can identify which message to remove from the data source.
4. DSE invokes the \texttt{rtl_store_flush_batch_end()} function to commit the batch of ticks to the store.

5. DSE invokes the \texttt{rtl_transport_source_batch_end()} function.
   - If step 4 succeeds, the batch end mode parameter is set to \texttt{RTL\_BATCH\_COMMIT}.
   - If step 4 fails, the batch end mode parameter is set to \texttt{RTL\_BATCH\_ROLLBACK}.

\textbf{Transport shutdown}

Transport shutdown occurs when DSE shuts down, and occurs once for each feed. After a transport is shut down, no other transport functions are invoked on the associated feed or on the flusher threads that are assigned to a feed.

The following flow for shutdown applies to each feed thread:

1. DSE invokes the \texttt{rtl_transport_target_close()} function.
   A pointer to the user data that was allocated by the \texttt{rtl_transport_init()} function is passed to the transport. The transport must not yet free the user data.

2. DSE invokes the \texttt{rtl_transport_source_close()} function.
   A pointer to the user data that was allocated by the \texttt{rtl_transport_init()} function is passed to the transport. The transport must not yet free the user data.

3. DSE invokes the \texttt{rtl_transport_shutdown()} function.
   A pointer to the user data that was allocated by the \texttt{rtl_transport_init()} function is passed to the transport. The transport must now free the user data.

   After the function returns, DSE nulls the pointer to the user data so it can no longer be accessed by the transport.

If DSE is running in restartable mode, the following flow for shutdown applies to each flusher thread:

1. DSE invokes the \texttt{rtl_transport_source_close()} function.
   A pointer to the user data that was allocated by the \texttt{rtl_transport_init()} function is passed to the transport. The transport must not yet free the user data.

2. DSE invokes the \texttt{rtl_transport_shutdown()} function.
   A pointer to the user data that was allocated by the \texttt{rtl_transport_init()} function is passed to the transport. The transport must now free the user data.

   After the function returns, DSE nulls the pointer to the user data so it can no longer be accessed by the transport.

\textbf{Transport handler shutdown}

Transport handler shutdown occurs only once for each invocation of DSE. After the transport handler is shut down, no other transport functions are invoked.

The following shutdown flow applies to each transport handler that is defined in the DSE configuration file:

1. DSE invokes the \texttt{rtl_transport_handler_shutdown} function.
   A pointer to the transport handler user data that was allocated by the \texttt{rtl_transport_handler_init} function is passed to the transport handler. The transport handler frees the user data.

   After the function returns, DSE nulls the pointer to the user data so that it can no longer be accessed by the transport handler.

2. DSE unloads the transport handler shared library.
**Error handling and logging for transport handlers**

DSE provides a mechanism for transport handlers to log messages into the DSE message log.

The logging mechanism for transport handlers is very similar to the logging mechanism for feed handlers. The only differences between them are:

- The transport handler uses a single logging function, rtl_transport_log().
- The letter in the log messages representing the component is T instead of F.

The recommendations for error handling in feed handlers also apply to transport handlers. See [“Error handling and logging for feed handlers” on page 158](#) for more information.

**Example:**

The message file contains the following entry:

```
123:E:Cannot open file "%s". Errno=%d
```

The following call to the rtl_transport_log() function logs the error:

```
rtl_transport_log(transport_id, 123, source_file_name, errno);
```

Where:

- The first parameter (transport_id) is the transport number that DSE passes into each transport interface function.
- The second parameter (123) is the message number to be logged.
- The remaining parameters are the variable arguments list to replace the printf style format string conversion specifications in the message text.
  - The source_file_name parameter is a character string variable.
  - The errno parameter is an integer variable.

The resulting log file output looks like this:

```
```

**Trace mechanism for transport handlers**

DSE provides a mechanism for transport handlers to add trace messages into the DSE trace output file.

The trace mechanism for transport handlers is almost identical to the one for feed handlers. The only difference between the mechanisms is the trace component definition. The following definition is for the transport handler trace mechanism:

```
#define COMPONENT COMPONENT_ID_TRANSPORT
```

See [“Trace mechanism for feed handlers” on page 159](#) for more information about the trace mechanism.

**Related concepts**

[Chapter 31, “DSE traces,” on page 485](#)

DSE provides a trace mechanism to aid in troubleshooting problems. Set up the trace file and settings using the trace configuration parameters.
Development and deployment of transport handlers

After you develop the transport, build and deploy it in your runtime environment.

DSE transport handlers are shared libraries that are loaded at runtime based on the configuration file. To develop your own transport handler, use the rtl_transport.h header file in the include directory as a starting point. To deploy your transport handler, you must configure DSE with the location of your transport handler shared library and your transport handler message file for log messages. These file locations are defined by the TRANSPORTXX_LIB_PATH and TRANSPORTXX_MSG_PATH configuration parameters.

See “Transport configuration parameters” on page 82 for more information.

Supplied transport handlers

You might not need to develop your own transport handler. DSE provides three pre-built transport handlers that are appropriate for many applications.

The shared libraries and message file are as follows:

DSE_INSTALL_DIR/lib/transpfile.so
A transport handler that reads messages from and writes messages to files.

DSE_INSTALL_DIR/lib/transpsock.so
A transport handler that reads messages from and writes messages to TCP/IP sockets.

DSE_INSTALL_DIR/lib/transpmsg.so
A transport handler that reads messages from and writes messages to WebSphere MQ queues.

DSE_INSTALL_DIR/msg/rtltransportmsgs.txt
A common message file for all of the supplied transport handlers.

The DSE_INSTALL_DIR directory is the directory in which DSE is installed.

Source and target names for pre-built transports

DSE provides pre-built transports for files, sockets, and queues. The source and target names must follow the appropriate format for the transport.

The pre-built transports are:

File transport
The pathname of the source or target data file. It can be either an absolute path or a path that is relative to the location of the DSE configuration file.

WebSphere MQ transport
"queue_manager:queue" The queue manager name is not required if it is the default queue manager. Both source and target queues must be on the same queue manager.

Example:
FEED01_SOURCE "venus.queue.manager:ORANGE.QUEUE"

Socket transport
"host:port" host can be either an IP address or the hostname of a machine. port can be either a port number of the name of an entry in the /etc/services file.
Examples:
FEED01_SOURCE "127.0.0.1:12345"
FEED01_SOURCE "127.0.0.1:dse_feed"
FEED01_SOURCE "remote.host.com:23456"

TRANSPORTXX_INFO parameter for pre-built transports

You can modify the behavior of each pre-built transport by specifying additional information in the TRANSPORTXX_INFO parameter.

The value of this parameter is a quoted string, containing list of name-value pairs separated by commas (such as "NAME1=value1, NAME2=value2"). Blank spaces before and after the names and values are optional.

The name-value pairs that the pre-built transports recognize are as follows:

File transport:

\texttt{READBUFSIZE=\textit{numbytes}}

Read data from the file in \textit{numbytes} chunks. If this value is not specified, each file read I/O call obtains the minimum number of bytes to obtain one complete message. Specify a larger number to significantly improve read performance.

Socket transport:

\texttt{SOCKBUFSIZE=\textit{numbytes}}

Set the size of the socket send and receive buffers to \textit{numbytes} bytes. The default buffer sizes are set by the operating system.

\texttt{CONNECT_RETRIES=\textit{count}}

Attempt to connect to a socket \textit{count} times, at one second intervals, before placing the feed in an idle state. The default retry count is 60.

\texttt{READBUFSIZE=\textit{numbytes}}

Read data from the socket in \textit{numbytes} chunks. If not specified, each file socket read I/O call obtains the minimum number of bytes to obtain one complete message. Specify a larger number to significantly improve read performance.

Websphere MQ transport:

\texttt{WAITINTERVAL=\textit{number}}

Wait up to \textit{number} milliseconds for a message to arrive on the source queue. If no message is available, close the queue and place the feed in an idle state. If number is -1, wait indefinitely for a message to arrive. The default wait time is 10 seconds.

Related reference

"TRANSPORTXX_INFO" on page 82
This optional parameter specifies additional information to be interpreted by the transport.

Restrictions for pre-built transports

The file and socket transports cannot be used in restartable mode (\texttt{RESTART_ENABLE=1}).
The socket transport participates in the socket connection sequence as a client application. It calls connect() to establish a connection to sources and targets. Data source and target applications must be started before DSE and must listen for connection requests.

The Websphere MQ transport does not use the start, end, separator, and length indicators in the rtl_tpt_msgfmt_t structure. Entire messages are read from source queues and are written to target queues, using MQGET and MQPUT function calls. Start, end, separator, and length indicators, if specified in the rtl_tpt_msgfmt_t transport message format data structures, are not removed from source messages and are not added to target messages.
Chapter 25. DSE interfaces

DSE defines interfaces that you must implement, for feed handlers, transports, storage systems, and high availability communications.

Related concepts

Chapter 21, “SQL queries with the DSE wrapper,” on page 137
You can query the data in DSE shared memory using SQL through the DSE wrapper. The wrapper enables you to create a nickname in the DB2 database system, which you can query using SQL to retrieve data from DSE. The DB2 database that is running the DSE wrapper can be on a remote system or on the same system as the DSE instance that is being queried.

Related tasks

Chapter 26, “Writing client applications,” on page 275
You can write C or Java client applications that connect directly to DSE to query and update data and retrieve statistics.

Feed handler interface

The feed handler defines the interface for the data feed, handling the feed data format and the processing that is required.

DSE provides four examples of feed handlers (included in the examples/generic, examples/ha, examples/simple, and examples/taq subdirectories.) You must implement your own feed handler that is customized for your data feed. You should work with an IBM consultant to build your feed handler.

The feed handler interface functions are defined in rtl_handler.h.

The interface consists of the following functions:

- rtl_handler_init()
- rtl_handler_shutdown()
- rtl_feed_init()
- rtl_feed_shutdown()
- rtl_feed_get_tick_from_message()
- rtl_feed_update_metadata()
- rtl_feed_get_publish_message()
- rtl_feed_get_message_fmts()

Related concepts

“Control flows for feed handlers” on page 156
All feed handler processing follows a set of control flows. Understand the flows so that you can develop your own feed handler. The flows use the feed handler interface.
\textbf{rtl_handler_init()}

This function is called by DSE to initialize a feed handler.

This function is called once for each feed handler and before any other feed handler function.

\textbf{Syntax}

\begin{verbatim}
int rtl_handler_init(
    int handler_id,
    void **handler_user_data);
\end{verbatim}

\textbf{Arguments}

\textbf{handler_id}

The ID of the feed handler that is being initialized.

\textbf{handler_user_data}

Return pointer to a user data area for keeping information about the feed handler. This function allocates and initializes whatever user data it requires and sets the \texttt{handler_user_data} pointer to point to it. The user data is available to other functions in the feed handler. The user data is deallocated by the \texttt{rtl_handler_shutdown} function.

\textbf{Return values}

If \texttt{rtl_handler_init()} is successful, it returns \texttt{RTL_SUCCESS}.

If \texttt{rtl_handler_init()} fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_handler_shutdown()**

This function is called as part of shutdown to close and free resources for a feed handler that is used as a feed handling mechanism for DSE.

This function is called once and is always called after the rtl_handler_init() function and any other feed handler function. After this shutdown call is made, no other feed handler calls are made for that feed handler.

**Syntax**

```c
int rtl_handler_shutdown(
    int handler_id,
    void *handler_user_data);
```

**Arguments**

**handler_id**

The ID of the feed handler that is being shut down.

**handler_user_data**

A user data area that is used for keeping information about the feed handler. This function deallocates this area and releases any other resources.

**Return values**

If rtl_handler_shutdown() is successful, it returns RTL_SUCCESS.

If rtl_handler_shutdown() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_feed_init()**

This function is called to initialize a feed.

This function is always called before any feed functions are called on the feed thread and after the feed handler has been initialized. There can be many feeds configured for DSE. Each feed executes in its own thread, and each thread calls this function on initialization.

**Syntax**

```
int rtl_feed_init(
    int feed_id,
    void **feed_user_data);
```

**Arguments**

- **feed_id**
  The ID of the feed that is being initialized.

- **feed_user_data**
  Return pointer to a user data area for keeping information about the feed. This function allocates and initializes whatever user data it requires and sets the feed_user_data pointer to point to it. The user data is available to other functions in the feed. The user data is deallocated by the rtl_feed_shutdown function.

**Return values**

If rtl_feed_init() is successful, it returns RTL_SUCCESS.

If rtl_feed_init() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_feed_shutdown()**

This function is called to shut down and free resources for a feed.

This function is always called after the `rtl_feed_init()` function and any feed functions. No feed functions are called on the feed thread after this function without calling the `rtl_feed_init()` function again. DSE can have many feeds. Each feed executes in its own thread, and each thread calls this function on shutdown.

**Syntax**

```c
int rtl_feed_shutdown(
    int feed_id,
    void *feed_user_data);
```

**Arguments**

- **feed_id**
  
  The ID of the feed that is being shut down.

- **feed_user_data**
  
  A user data area that is used for keeping information about the feed. This function deallocates this area and releases any other resources.

**Return values**

If `rtl_feed_shutdown()` is successful, it returns RTL_SUCCESS.

If `rtl_feed_shutdown()` fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_feed_get_tick_from_message()**

This function parses messages from the feed data source.

DSE calls the `rtl_feed_get_tick_from_message()` function after calling the `rtl_transport_source_get_message()` function. The `rtl_feed_get_tick_from_message()` function can be called repeatedly during the operation of a feed.

**Syntax**

```c
int rtl_feed_get_tick_from_message(
    int feed_id,
    void *feed_user_data,
    char *msg_buf,
    int msg_len,
    char *symbol,
    void *tick,
    char *nulls,
    int *tick_store_id,
    int *metadata_store_id,
    int iteration);
```

**Arguments**

- **feed_id (in)**  
  The ID of the feed.

- **feed_user_data (in)**  
  The user data area allocated by the `rtl_feed_init()` function for keeping information about the feed.

- **msg_buf (in)**  
  A pointer to the message obtained from the data source by the `rtl_transport_source_get_message()` function.

- **msg_len (in)**  
  The length of the message data in the message buffer.

- **symbol (out)**  
  The symbol that corresponds to the tick. This argument is ignored if the `tick_store_id` argument is set to the store ID of a simple schema store.

- **tick (out)**  
  The tick read from the feed data source.

- **nulls (out)**  
  A null indicators array (one byte per tick field). This array is allocated by DSE. The feed handler should set each value in the array to either `RTL_FIELD_IS_NOT_NULL` or `RTL_FIELD_IS_NULL` accordingly.

- **tick_store_id (out)**  
  The ID of the store in which the tick should be stored. A value of -1 indicates that the tick should not be stored.

- **metadata_store_id (out)**  
  The ID of the store of which the tick updates the metadata. A value of -1 indicates that metadata should not be updated.

  This ID cannot be the store ID for a simple schema store. If it is:
  - A warning message is logged.
  - A value of -1 is assumed for this argument, and DSE does not call the `rtl_feed_update_metadata()` function.
iteration (in)
A counter that is supplied and maintained by DSE that the
rtl_feed_get_tick_from_message() function can use for keeping track of the
number of ticks in a message that contains more than one tick. DSE
initializes iteration to 0 and calls the rtl_feed_get_tick_from_message()
function repeatedly, incrementing iteration by 1, until it returns
RTL_NO_MORE_TICKS.

Return values
RTL_MORE_TICKS
If symbol, tick, and nulls contain information for a valid tick. DSE calls the
rtl_feed_get_tick_from_message() function again to get the next tick from
the message.

RTL_LAST_TICK
If symbol, tick, and nulls represent the last tick to be obtained from the
message. This is an optimization so that DSE only needs to call the
rtl_feed_get_tick_from_message() function once for the typical case in
which there is only one tick per message.

RTL_NO_TICK
If symbol, tick, and nulls do not contain a valid tick. DSE does not call the
rtl_feed_get_tick_from_message() function again until it gets another
message from the data source.

RTL_FAILURE
If there are any problems.
**rtl_feed_update_metadata()**

This function is called to update the metadata for a symbol.

This function is always called after calling the rtl_feed_get_tick_from_message() function. This function can be called repeatedly during a feed’s operation but cannot be called after the rtl_transport_source_close() function is called unless the rtl_transport_source_open() function is called again.

**Syntax**

```c
int rtl_feed_update_metadata(
    int feed_id,
    void *feed_user_data,
    int synced,
    char *msg_buf,
    int msg_len,
    char *symbol,
    void *tick,
    char *tick_nulls,
    void *sechdr,
    char *sechdr_nulls,
    int *flags);
```

**Arguments**

- **feed_id** *(in)*
  The ID of the feed.

- **feed_user_data** *(in)*
  The user data area allocated by the rtl_feed_init() function for keeping information about the feed.

- **synced** *(in)*
  Boolean flag indicating if the metadata being updated is synchronized. This can only be false during synchronization in a high availability cluster configuration.

- **msg_buf** *(in)*
  A pointer to the message obtained form the data source by the rtl_transport_source_get_message() function.

- **msg_len** *(in)*
  The length of the message data in the message buffer.

- **symbol** *(in)*
  The symbol that the metadata is being updated for.

- **tick** *(in)*
  The tick read from the feed data source.

- **tick_nulls** *(in)*
  A tick data null indicators array (one byte per tick field).

- **sechdr** *(in/out)*
  The current metadata area for the symbol to be updated.

- **sechdr_nulls** *(in/out)*
  A symbol metadata null indicators array (one byte per metadata field). This array is allocated by DSE. The feed handler should set each value in the array to either RTL_FIELD_IS_NOT_NULL or RTL_FIELD_IS_NULL accordingly.

- **flags** *(in/out)*
  A bit mask to indicate if any fields in the metadata that are persisted to the...
store are changed. If a field that is configured as READWRITE is changed, the RTL_META_RW_CHANGED bit should be set. If a field that is configured as READWRITE AUTO is changed, the RTL_META_RW_AUTO_CHANGED bit should be set.

Return values

If rtl_feed_update_metadata() is successful, it returns RTL_SUCCESS.

If rtl_feed_update_metadata() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
rtl_feed_get_publish_message()

This function is called to determine if a message should be published to the feed data target.

This function is always called after a tick is read or received from the feed data source and metadata updated on the feed thread. This function can be called repeatedly during a feed’s operation but cannot be called after the rtl_transport_source_close() function is called unless the rtl_transport_source_open() function is called again.

Syntax

```c
int rtl_feed_get_publish_message(
    int feed_id,
    void *feed_user_data,
    char *symbol,
    void *tick,
    char *tick_nulls,
    void *sechdr,
    char *sechdr_nulls,
    char **msg_buf,
    int *msg_len);
```

Arguments

**feed_id (in)**
- The ID of the feed.

**feed_user_data (in)**
- The user data area allocated by the rtl_feed_init() function for keeping information about the feed.

**symbol (in)**
- The symbol that corresponds to the metadata and tick.

**tick (in)**
- The tick read from the feed data source.

**tick_nulls (in)**
- A tick data null indicators array (one byte per tick field).

**sechdr (in)**
- The metadata for the symbol.

**sechdr_nulls (in)**
- A symbol metadata null indicators array (one byte per metadata field).

**msg_buf (out)**
- A return pointer to the message to be published to the feed data target. This buffer is passed to the rtl_transport_put_message() function. It is the feed handler’s responsibility to allocate and free the buffer, and it is recommended that this buffer be cached in a user data area and reused for performance.

**msg_len (out)**
- The length of the message data in the message buffer. If the length is less than or equal to 0, then no message is published.

Return values

If rtl_feed_get_publish_message() is successful, it returns RTL_SUCCESS.
If `rtl_feed_get_publish_message()` fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_feed_get_message_fmts()**

The `rtl_feed_get_message_fmts()` function is called during feed initialization to get information about the basic structure of the messages to be read from a data source and written to a data target. (Incoming and outgoing messages do not need to have the same format.)

The feed thread uses this function to convey message structure information from the feed handler to the transport.

The `rtl_feed_get_message_fmts()` function is called after the `rtl_feed_init()` function and before the `rtl_transport_init()` function.

**Syntax**

```c
int rtl_feed_get_message_fmts(
    int feed_id,
    void *feed_user_data,
    rtl_tpt_msgfmt_t *source_msgfmt,
    rtl_tpt_msgfmt_t *target_msgfmt);
```

**Arguments**

- **feed_id (in)**
  - The ID of the feed.

- **feed_user_data (in)**
  - A user data area allocated by the `rtl_feed_init()` function for storing information about the feed.

- **source_msgfmt (out)**
  - A pointer to a data structure that describes the basic structure of a source message. The data structure must be populated by this function. See `rtl_tpt_msgfmt_t` data structure for more information.

- **target_msgfmt (out)**
  - A pointer to a data structure that describes the basic structure of a target message. The data structure must be populated by this function. See “`rtl_tpt_msgfmt_t` data structure” on page 253 for more information.

**Return values**

If `rtl_feed_get_message_fmts()` is successful, it returns RTL_SUCCESS.

If `rtl_feed_get_message_fmts()` fails, it returns RTL_FAILURE.
Generic storage support interfaces

DSE defines generic interfaces for receiving data from a backing store and persisting data to a backing store. DSE uses these interfaces to communicate with a storage system.

DSE provides two implementations of these interfaces, for:
- The DB2 database system
- A test storage system
  This system does not store any data. Use it to test DSE in a configuration where data is not persisted to a storage system. The source code for this implementation is also included in the storesys/test subdirectory.
  The source code for the test storage system is provided.

To enable error recovery for storage systems, as described in Chapter 19, “Enabling error recovery for storage systems,” on page 131, ensure that your storage system implementation:
- Identifies the specific error conditions from which recovery is possible.
- Reports them to DSE using the appropriate return codes.

Storage system interface

DSE uses this interface to open and close a storage system. The storage system interface functions are defined in rtl_gen_store.h.

The storage system interface consists of the following functions:
- rtl_storesys_close()
- rtl_storesys_init()
- rtl_storesys_open()
- rtl_storesys_shutdown()
rtl_storesys_close()
This function is called as part of shutdown to close and free resources for a storage system used as a storage mechanism for DSE.

This function is called once for each thread which uses the storage system and is always called after the rtl_store_sys_open() function and any other storage mechanism function operating on the store. After this close call is made, no other store calls will be made for that storage system on that thread, the rtl_storesys_shutdown() function will be called on a global level to shut down the storage system after all storage systems have been closed for all threads accessing them.

Syntax

int rtl_storesys_close(
    void *storesys,
    void *sess_user_data
    rtl_connect_mode_t connect_mode;

Arguments

storesys
The descriptor of the storage system which is being opened. Can be used in conjunction with storage system descriptor accessor functions to retrieve information about the storage system from the configuration file.

sess_user_data
A user data area for keeping track of information about the storage system. This function deallocates this area and releases any other resources.

connect_mode
An input parameter to indicate whether the function is called to close the storage system normally, or to close it in a way that allows it to be reopened later. If connect_mode is set to RTL_STORE_CONNECT, this function closes the storage system and releases all of the previously allocated resources. If connect_mode is set to RTL_STORE_RECONNECT, this function closes the storage system in a way that allows it to be reopened later.

Return values

If rtl_storesys_close() is successful, it returns RTL_SUCCESS.

If rtl_storesys_close() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
rtl_storesys_init()
This function is called as part of initialization to initialize a storage system for use as a storage mechanism for DSE.

This function is called once for each storage system and is always called before any other storage system function operating on the storage system.

Syntax
```c
int rtl_storesys_init(
    int storesys_id,
    void **storesys_user_data);
```

Arguments
storesys_id
The ID of the storage system which is being opened. This id can be used in conjunction with storage system descriptor accessor functions to retrieve information about the storage system from the configuration file.

storesys_user_data
A return pointer to a user data area for keeping information about the storage system. This function allocates and initializes whatever user data it requires and sets the storesys_user_data pointer to point to it. The user data is available to other functions that access the storage system. The user data is deallocated by the rtl_storesys_shutdown() function.

Return values
If rtl_storesys_init() is successful, it returns RTL_SUCCESS.

If rtl_storesys_init() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_storesys_open()**

This function is called as part of initialization to open a storage system for use as a storage mechanism for DSE.

This function is called once for each thread that uses the storage system. For example, a storage system with 3 stores, each configured with 5 flushers, calls the rtl_storesys_open() function 15 times for the flushers (once for each flusher). Each flusher has its own sess_user data which is available to subsequent store calls on the same thread. This function is always called before the rtl_storesys_close() function and any store function operating on the store.

**Syntax**

```c
int rtl_storesys_open(
    void *storesys,
    void **sess_user_data,
    rtl_store_connect_mode_t connect_mode,
    rtl_store_tx_mode_t tx_mode);
```

**Arguments**

- `storesys`:
  The descriptor of the storage system which is being opened. Can be used in conjunction with storage system descriptor accessor functions to retrieve information about the storage system from the configuration file.

- `sess_user_data`:
  A return pointer to a user data area for keeping information about the storage system. This area is allocated and initialized by this function. This area is used by functions that access the storage system on the same thread. This area is deallocated by the rtl_storesys_close() function.

- `connect_mode`:
  An input parameter to indicate whether the function is called to open the storage system for the first time, or to reopen the storage system after it has been closed. If `connect_mode` is set to RTL_STORE_CONNECT, this function opens the storage system and allocates the required resources. If `connect_mode` is set to RTL_STORE_RECONNECT, this function reopen the storage system that was previously closed in a way that allows it to be reopened later.

- `tx_mode`:
  An input parameter to indicate whether the rtl_store_flush_batch_end() function can commit or rollback a transaction. Valid values are:

  **RTL_STORE_TXSTORESYS**
  Enables commits or rollbacks by the rtl_store_flush_batch_end() function.

  **RTL_STORE_TXEXTERNAL**
  Disables commits or rollbacks by the rtl_store_flush_batch_end() function.
Return values

The rtl_storesys_open() function returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_SUCCESS</td>
<td>The function opened the storage system successfully.</td>
</tr>
<tr>
<td>RTL_FAILURE</td>
<td>The function failed to open the storage system.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
rtl_storesys_shutdown()
This function is called as part of shutdown to close and free resources for a storage system used as a storage mechanism for DSE.

This function is called once and is always called after rtl_storesys_init() and any other storage mechanism function operating on the store. After this close call has been made no other storage system calls will be made for that storage system.

Syntax
```c
int rtl_storesys_shutdown(
    int storesys_id,
    void *storesys_user_data);
```

Arguments
- **storesys_id**
  - The ID of the storage system which is being opened. This id can be used in conjunction with storage system descriptor accessor functions to retrieve information about the storage system from the configuration file.

- **storesys_user_data**
  - A user data area that is used for keeping information about the storage system. This function deallocates this area and releases any other resources.

Return values
If rtl_storesys_shutdown() is successful, it returns RTL_SUCCESS.

If rtl_storesys_shutdown() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
Store interface

The store interface supports a generic store of ticks and symbols. DSE uses this interface to open and close a store.

The interface consists of the following functions:

- rtl_store_close()
- rtl_store_open()

Related concepts

“Store API” on page 289
DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_close()**

This function is called to close a store.

The main shutdown code will call it as well as each individual flusher and inserter thread as it is shut down. It is always called after the rtl_store_open() function and any symbol or flush store API calls. No store calls will be made on the store after this function without calling the rtl_store_open() function again.

**Syntax**

```c
int rtl_store_close(
    void *store,
    void *store_user_data,
    rtl_store_mode_t store_mode,
    rtl_connect_mode_t connect_mode);
```

**Arguments**

store  
The descriptor of the store which is being opened. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

store_user_data  
A user data area for keeping track of information about the store. This function deallocates this area and releases any other resources.

store_mode  
A store can be closed in several modes. You must call the rtl_store_close function with the same store_mode with which you called the rtl_store_open() function. The modes are:

- RTL_STORE_PARTITION_TICK_MODE
- RTL_STORE_SYM_ADD_MODE
- RTL_STORE_SYM_DEL_MODE
- RTL_STORE_SYM_SCAN_MODE
- RTL_STORE_SYM_UPD_AUTO_MODE
- RTL_STORE_SYM_UPD_MAN_MODE
- RTL_STORE_TICK_FLUSH_MODE
- RTL_STORE_TICK_FLUSH_SIMPLE_MODE
- RTL_STORE_TICK_SCAN_MODE

For a description of these modes, see ["rtl_store_open()" on page 195](#).

connect_mode  
An input parameter to indicate whether the function is called to close the store normally, or to close it in a way that allows it to be reopened later. If connect_mode is set to RTL_STORE_CONNECT, this function closes the store and releases all of the previously allocated resources. If connect_mode is set to RTL_STORE_RECONNECT, this function closes the store in a way that allows it to be reopened later.

**Return values**

If rtl_store_close() is successful, it returns RTL_SUCCESS.

If rtl_store_close() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
rtl_store_open()
This function is called to initialize a store.

It is always called before any symbol or store API calls and after the storage system has been opened on this thread by the rtl_storesys_open() function. A store can be used by several parts of DSE in several different modes. The main startup code accesses a store to scan all the symbols from the store to initialize the sec_hdr pool in memory. Each flusher thread accesses a store to write ticks to the store from memory. Each inserter thread accesses a store to write new symbols to the store. A store can also be accessed by the listener thread for deleting symbols as part of the purging of old symbols functionality.

Syntax
int rtl_store_open(
    void *store,
    void **store_user_data,
    rtl_store_mode_t store_mode,
    int id,
    rtl_connect_mode_t connect_mode);

Arguments
store The descriptor of the store which is being opened. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

store_user_data A return pointer to a user data area for keeping track of information about the store. This area is allocated and initialized by this function. Each store function can access this area through accessor functions to maintain state. This area is deallocated by the rtl_store_close() function.

store_mode A store can be opened in several modes relating to how it is accessed. The modes are as follows:

RTL_STORE_PARTITION_TICK_MODE
    For the partitioning threads that process ticks data. Only the partitioning functions are called for this store mode.

RTL_STORE_SYM_SCAN_MODE
    For the main startup code that reads symbols from the store. Only the symbol scan and flusher allocation functions are called for this store mode.

RTL_STORE_SYM_ADD_MODE
    For the inserter threads that write new symbols to the store. Only the symbol add and flusher allocation functions are called for this store mode.

RTL_STORE_SYM_DEL_MODE
    For the listener threads that delete old symbols from the store. Only the symbol delete function are called for this store mode. This mode is not yet implemented.

RTL_STORE_SYM_UPD_AUTO_MODE
    For the metadata persistence threads that write automatically persisted metadata to the store. Only the symbol update store functions are called for this store mode.
RTL_STORE_SYM_UPD_MAN_MODE
For the metadata persistence threads that write manually persisted metadata to the store. Only the symbol update store functions are called for this store mode.

RTL_STORE_TICK_FLUSH_MODE
For the flusher threads that write ticks to the store. Only the tick flushing functions are called for this store mode.

RTL_STORE_TICK_FLUSH_SIMPLE_MODE
For the flusher threads that write ticks to simple schema stores.

RTL_STORE_TICK_SCAN_MODE
For the scanner threads that read ticks from the store. Only the tick scanning functions are called for this store mode.

id
Set if the store is being open by a flusher thread otherwise it is undefined. In the case of a flusher thread the id holds the flusher number.

connect_mode
An input parameter to indicate whether the function is called to open the store for the first time, or to reopen the store after it has been closed. If connect_mode is set to RTL_STORECONNECT, this function opens the store and allocates the required resources. If connect_mode is set to RTL_STORERECONNECT, this function reopen the store that was previously closed in a way that allows it to be reopened later.

Return values
If rtl_store_open() is successful, it returns RTL_SUCCESS.

If rtl_store_open() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
Partitioning interface

DSE uses this interface to support data partitioning in the store.

The interface consists of the following function:

- rtl_store_get_partition_mode()
- rtl_store_symbol_allocate_flusher()
- rtl_store_tick_allocate_flusher()
**rtl_store_get_partition_mode()**

This function indicates the partitioning method that the store is using.

DSE calls this function on startup after calling the rtl_store_open() function.

**Syntax**

```c
rtl_store_partition_mode_t rtl_store_get_partition_mode(void *store)
```

**Arguments**

- **store**  
  The descriptor of the store. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

**Return values**

If `rtl_store_get_partition_mode()` is successful, it returns the partition mode of the storage system. The partition modes are:

- **RTL_TICK_PARTITION**  
  The store is partitioning data on a per-tick basis.

- **RTL_SYMBOL_PARTITION**  
  The store is partitioning data on a per-symbol basis.

- **RTL_NO_PARTITION**  
  The store is not partitioning data.

If `rtl_store_get_partition_mode()` fails, it returns `RTL_FAILURE`. 
**rtl_store_symbol_allocate_flusher()**

This function assigns symbols to a flusher when the store partition mode is RTL_NO_PARTITION or RTL_SYMBOL_PARTITION.

It is called once for each symbol when either of the following events occur:
- All existing symbols are read from the store by DSE during its initialization.
- New symbols are added to the store.

The rtl_store_symbol_allocate_flusher() function is called after the rtl_store_open() function.

**Syntax**

```c
int rtl_store_symbol_allocate_flusher(
    void *store,
    int num_flushers,
    int symbol_id,
    char *symbol_name);
```

**Arguments**

- **store** 
  The descriptor of the store which is being scanned for symbols or to which symbols are being added. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **num_flushers** 
  The number of flushers that are configured for the store.

- **symbol_id** 
  The ID of the symbol to be assigned a flusher.

- **symbol_name** 
  The name of the symbol to be assigned a flusher. This string is defined to be a NULL terminated string of length MAX_SYMBOL_LEN (including the NULL).

**Return values**

If rtl_store_symbol_allocate_flusher() is successful, it returns the number of the flusher to which this symbol should be assigned.

If rtl_store_symbol_allocate_flusher() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>* The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>* The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
</tbody>
</table>

Chapter 25. DSE interfaces   199
<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_tick_allocate_flusher()**

DSE calls this function to allocate a tick to a flusher.

The partitioning threads call this function after the rtl_store_open() function is called. It is called only if the store is partitioning data on a per-tick basis.

**Syntax**

```c
int rtl_store_tickAllocateFlusher(
    void *store,
    int num_flushers,
    int symbol_id,
    char *symbol_name,
    void *tick,
    char *nulls,
    int num_fields);
```

**Arguments**

- **store**
  The descriptor of the store for which the ticks are being partitioned. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **num_flushers**
  The number of flushers that are configured for the store.

- **symbol_id**
  The ID of the symbol whose tick is being flushed to the store.

- **symbol_name**
  The name of the symbol whose tick is being flushed to the store. This string is defined to be a NULL-terminated string of length MAX_SYMBOL_LEN (including the NULL).

- **tick**
  The tick to be stored.

- **nulls**
  A zero-based array indicating which fields in the tick are null.

  **RTL_FIELD_IS_NULL**
  Indicates that the field is null.

  **RTL_FIELD_IS_NOT_NULL**
  Indicates that the field holds a non-null value.

- **num_fields**
  The number of fields in the tick.

**Return values**

If rtl_store_tickAllocateFlusher() is successful, it returns the number of the flusher to which the tick should be assigned.

If rtl_store_tickAllocateFlusher() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>Return code</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
Symbol adding interface

DSE uses this interface to add new symbols and their metadata to the store.

For DSE to write symbol metadata to the store, the following conditions must be met:

- The STORExx_META_MODE configuration file parameter must be non-zero.
- The RECORDXX_META_SCHEMA configuration parameter must designate one or more metadata fields as STORE READ, STORE READWRITE, or STORE READWRITE AUTO.

If both of these conditions are not met, only symbol names are written to the store.

The interface consists of the following functions:

- rtl_store_symbol_add()
- rtl_store_symbol_add_batch_begin()
- rtl_store_symbol_add_batch_end()
- rtl_store_symbol_add_begin()
- rtl_store_symbol_add_end()
- rtl_store_symbol_getid()
**rtl_store_symbol_add()**

This function is called for each new symbol to be added to the store.

This function is always called between pairs of calls to the
rtl_store_symbol_add_batch_begin() and rtl_store_symbol_add_batch_end() functions.

**Syntax**

```c
int rtl_store_symbol_add(
    void *store,
    int *symbol_id,
    char *symbol_name,
    void *metadata,
    char *nulls,
    int num_fields,
    void *sym_add_user_data);
```

**Arguments**

- **store**  The descriptor of the store to which symbols are added. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.
- **symbol_id**  The ID of the symbol allocated by the store and returned through this parameter.
- **symbol_name**  The name of the symbol to be added. This string is defined to be a NULL terminated string of length MAX_SYMBOL_LEN (including the NULL).
- **metadata**  A symbol metadata structure representing all of the fields that are defined by the RECORDXX_META_SCHEMA configuration parameter that is associated with the store. This structure must be allocated by the caller. This function writes to the store only the fields that are designated as STORE READ, STORE READWRITE, or STORE READWRITE AUTO in the RECORDXX_META_SCHEMA configuration parameter.
- **nulls**  A null indicators array (one byte per metadata field), indicating which metadata fields are set to null in the store.
- **num_fields**  The number of fields in the metadata.
- **sym_add_user_data**  A user data area for keeping state across the symbol add chain. This area is allocated and initialized by the rtl_store_symbol_add_begin() function. This area is passed to each symbol add function call to maintain state. This area is deallocated by the rtl_store_symbol_add_end() function.

**Return values**

If rtl_store_symbol_add() is successful, it returns RTL_SUCCESS.
If rtl_store_symbol_add() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_DUPLICATE_KEY</td>
<td>An attempt to insert a symbol in the store violates unique index rules. This code is returned if high availability is enabled with shared stores. If high availability is not enabled, or it is enabled without shared stores, RTL_FAILURE is returned.</td>
</tr>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried. This return code is returned if high availability is not enabled, or high availability is enabled without shared databases. If high availability is enabled with shared databases, RTL_DUPLICATE_KEY is returned.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_symbol_add_batch_begin()**
This function can be called multiple times during symbol addition; each call writes
a batch of new symbols and their metadata to the store.

A set of symbol additions can be broken down into several batches, as defined by
the STOREXX_INSERT_BATCH_SIZE parameter. Defining batches helps to prevent possible problems with writing a large amount of data to a store within a single transaction.

This function is always called before any symbols are added, and after the
rtl_store_symbol_add_begin() function is called.

**Syntax**

```c
int rtl_store_symbol_add_batch_begin(
    void *store,
    void *sym_add_user_data);
```

**Arguments**

- **store** The descriptor of the store to which symbols are being added. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **sym_add_user_data**
  A user data area for keeping state across the symbol add chain. This area is allocated and initialized by the rtl_store_symbol_add_begin() function. This area is passed to each symbol addition function call to maintain state. This area is deallocated by the rtl_store_symbol_add_end() function.

**Return values**

If rtl_store_symbol_add_batch_begin() is successful, it returns RTL_SUCCESS.

If rtl_store_symbol_add_batch_begin() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_store_symbol_add_batch_end()**

This function is called to complete each batch of new symbol additions.

A set of symbol additions can be broken down into several batches, as defined by the STOREXX_INSERT_BATCH_SIZE parameter. Defining batches helps to prevent possible problems with writing a large amount of data to a store within a single transaction.

This function can be called multiple times during symbol addition (once for each batch). This function is called after the rtl_store_symbol_add_batch_begin() function is called for the batch. No additional symbols are written until the rtl_store_symbol_add_batch_begin() function is called again.

If the function is called in commit mode, the store makes all of the changes for the batch permanent.

If the function is called in rollback mode, the store rolls back the changes.

If the rtl_store_symbol_add_batch_end() function is called in commit mode and returns a failure, the function is called again in rollback mode for the same batch.

**Syntax**

```c
int rtl_store_symbol_add_batch_end(
    void *store,
    rtl_batch_end_mode_t mode,
    void *sym_add_user_data);
```

**Arguments**

- **store**  
  The descriptor of the store to which symbols are added. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **mode**  
  The mode for ending the batch. If the batch is written with no errors, the store is asked to commit the batch. Otherwise, the store is asked to roll back the batch. The modes are:

  - **RTL_BATCH_COMMIT**  
    Commits the new symbols in the batch to the store.

  - **RTL_STORE_ROLLBACK**  
    Rolls back the new symbols in the batch from the store.

- **sym_add_user_data**  
  A user data area for keeping state across the symbol add chain. This area is allocated and initialized by the rtl_store_symbol_add_begin() function. This area is passed to each symbol add function call to maintain state. This area is deallocated by the rtl_store_symbol_add_end() function.

**Return values**

If rtl_store_symbol_add_batch_end() is successful, it returns RTL_SUCCESS.

If rtl_store_symbol_add_batch_end() fails, it returns one of the following return codes:
Table 22. Return codes

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
</tbody>
</table>
| RTL_FAILURE_RECONNECT | The storage system encountered an error code that indicates one of the following conditions:  
|                       | - The connection to the data store is lost.  
|                       | - The connection still exists, but previously established resources for the connection are broken.  
|                       | Operations cannot proceed until the connection is restored.        |
| RTL_FAILURE_RETRY     | The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried. |
**rtl_store_symbol_add_begin()**
This function is called at the beginning of each set of symbols to be added to the store.

This function must be called before any other symbol add function and after the rtl_store_open() function.

**Syntax**
```c
int rtl_store_symbol_add_begin(
    void *store,
    void **sym_add_user_data);
```

**Arguments**
- **store**   The descriptor of the store to which symbols are added. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.
- **sym_add_user_data** A return pointer to a user data area for keeping state across the symbol add chain. This area is allocated and initialized by this function. This area is passed to each symbol add function call to maintain state. This area is deallocated by the rtl_store_symbol_add_end() function.

**Return values**
If rtl_store_symbol_add_begin() is successful, it returns RTL_SUCCESS.

If rtl_store_symbol_add_begin() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_store_symbol_add_end()**

This function is called as part of the end of the addition of symbols to the store.

This function is called after a set of symbols are added and is always called after the `rtl_store_symbol_add_begin()` function. Any symbol addition functions that are called after this function is called must call the `rtl_store_symbol_add_begin()` function first.

**Syntax**

```c
int rtl_store_symbol_add_end(
    void *store,
    void *sym_add_user_data);
```

**Arguments**

- **store**  The descriptor of the store to which symbols are added. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **sym_add_user_data**  A user data area for keeping state across the symbol add chain. This function deallocates this area and releases any other resources.

**Return values**

If `rtl_store_symbol_add_end()` is successful, it returns RTL_SUCCESS.

If `rtl_store_symbol_add_end()` fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_store_symbol_getid()**

This function returns the symbol_id value for symbol that is specified by the symbol_name argument in the store.

This function is called when the rtl_store_symbol_add() function returns a RTL_DUPLICATE_KEY error.

**Syntax**

```c
int rtl_store_symbol_getid(
    void *store,
    int *symbol_id,
    char *symbol_name);
```

**Arguments**

- **store (in)**
  - The descriptor of the store that is being scanned for symbols. This argument can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **symbol_id (out)**
  - The ID of the symbol that is being returned.

- **symbol_name (in)**
  - The name of the symbol.

**Return values**

If rtl_store_symbol_getid() is successful, it returns RTL_SUCCESS and sets the value of symbol_id to the symbol id number for the symbol that is specified by the symbol_name argument in the store.

If rtl_store_symbol_getid() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
</tbody>
</table>
| RTL_FAILURE_RECONNECT | The storage system encountered an error code that indicates one of the following conditions:  
• The connection to the data store is lost.  
• The connection still exists, but previously established resources for the connection are broken. Operations cannot proceed until the connection is restored. |
| RTL_FAILURE_RETRY | The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried. |
Symbol scanning interface

The interface supports the generic reading of symbols and symbol metadata from a store. During its initialization, DSE uses this interface to read all symbols and symbol metadata from the store.

For DSE to read symbol metadata from the store during initialization, the following conditions must be met:

- The STOREXX_META_MODE configuration parameter must be non-zero.
- The RECORDXX_META_SCHEMA configuration parameter must designate one or more metadata fields as STORE READ, STORE READWRITE, or STORE READWRITE AUTO.

If both of these conditions are not met, only symbol names and IDs are read from the store at startup.

The interface consists of the following functions:

- rtl_store_symbol_begin_scan()
- rtl_store_symbol_end_scan()
- rtl_store_symbol_get_next()
**rtl_store_symbol_begin_scan()**

This function is called before scanning the store for all symbols.

This function is called once for each store. It is always called before the `store_symbol_get_next()` function and after the `rtl_store_open()` function.

**Syntax**

```c
int rtl_store_symbol_begin_scan(
    void *store,
    void **scan_user_data);
```

**Arguments**

- **store**  
  The descriptor of the store which is being scanned for symbols. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **scan_user_data**  
  A return pointer to a user data area for keeping state across the symbol scan chain. This area is allocated and initialized by this function. This area is passed to each call to the `rtl_store_symbol_get_next()` function to maintain state. This area is deallocated by the `rtl_store_symbol_end_scan()` function.

**Return values**

If `rtl_store_symbol_begin_scan()` is successful, it returns RTL_SUCCESS.

If `rtl_store_symbol_begin_scan()` fails, it returns one of the following return codes:

**Table 24. Return codes**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_symbol_end_scan()**
This function is called as part of termination after the store is scanned for all symbols.

This function is called once for each store and is always called after the rtl_store_symbol_begin_scan function. To call the rtl_store_symbol_get_next() function after this function, the rtl_store_symbol_begin_scan() function must be called first.

**Syntax**

```c
int rtl_store_symbol_end_scan(
    void *store,
    void *scan_user_data);
```

**Arguments**

- **store**  The descriptor of the store which is being scanned for symbols. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **scan_user_data**  A user data area for keeping state across the symbol scan chain. This function deallocates this area and releases any other resources.

**Return values**

If rtl_store_symbol_end_scan() is successful, it returns RTL_SUCCESS.

If rtl_store_symbol_end_scan() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
rtl_store_symbol_get_next()
This function is called repeatedly to retrieve all symbols for the store.

This function is always called after the rtl_store_symbol_begin_scan() function and before the rtl_store_symbol_end_scan() function. The rtl_store_symbol_get_next() function is called repeatedly until it indicates that there are no more symbols to return; no symbols are returned for the final invocation of this function.

Syntax

```c
int rtl_store_symbol_get_next(
    void *store,
    int *symbol_id,
    char *symbol_name,
    void *metadata,
    char *nulls,
    int *numfields,
    void *scan_user_data);
```

Arguments

store The descriptor of the store which is being scanned for symbols. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

symbol_id The ID of the next symbol being returned.

symbol_name The name of the next symbol being returned. The rtl_store_symbol_get_next() function must fill in the symbol_name argument with the name of the next symbol. This string is defined to be a NULL terminated string of length MAX_SYMBOL_LEN (including the NULL).

metadata A symbol metadata structure representing all of the fields that are defined by the RECORDXX_META_SCHEMA configuration parameter that is associated with the store. This structure must be allocated by the caller. This function initializes only the fields that are designated as STORE READ, STORE READWRITE, or STORE READWRITE AUTO in the RECORDXX_META_SCHEMA configuration parameter. All other fields are unchanged.

nulls A null indicators array. This array must be allocated by the caller and it must be long enough to hold null indicators (one byte per field) for all of the metadata fields. This function initializes only the null indicators for fields that are designated as STORE READ, STORE READWRITE, or STORE READWRITE AUTO in the RECORDXX_META_SCHEMA configuration parameter. Null indicators for all other fields are unchanged.

numfields The number of fields in the metadata.

scan_user_data A user data area for keeping state across the symbol scan chain. This area is allocated and initialized by the rtl_store_symbol_begin_scan() function. This area is deallocated by the rtl_store_symbol_end_scan() function.

Return values

If rtl_store_symbol_get_next() is successful, it returns RTL_SUCCESS.
If there are no more symbols to return, rtl_store_symbol_get_next() returns RTL_NO_MORE_RESULTS.

If rtl_store_symbol_get_next() fails, it returns one of the following return codes:

Table 26. Return codes

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
Symbol updating interface

DSE uses this interface to update symbol metadata in a store.

The interface consists of the following functions:

- rtl_store_symbol_set_update()
- rtl_store_symbol_update_batch_begin()
- rtl_store_symbol_update_batch_end()
- rtl_store_symbol_update_begin()
- rtl_store_symbol_update_end()
**rtl_store_symbol_set_update()**

This function is called for each set of symbol metadata that is written to the store. A set of symbol metadata can contain the metadata for many symbols.

This function is called many times while a set of metadata is written to the store.

**Syntax**

```c
int rtl_store_symbol_set_update(
    void *store,
    void **metadata_set,
    unsigned char **nulls_set,
    int num_fields,
    int *symbol_id_set,
    char **symbol_name_set,
    uint16 *status_set,
    int set_size,
    void *sym_update_user_data);
```

**Arguments**

- **store**: The descriptor of the store to which metadata is written.
- **metadata_set**: An array of metadata in shared memory.
- **nulls_set**: An array of bitwise null indicators.
- **num_fields**: The number of fields in the tick.
- **symbol_id_set**: An array of IDs of the symbols whose metadata is to be written to the store.
- **symbol_name_set**: An array of names of the symbol whose metadata is to be written to the store.
- **status_set**: An array of status indicators for the unflushed data rows. A particular row returns:
  - 0: If the row status is successful (status_set[i] = 0).
  - Non-zero: If the row status is not successful.
- **set_size**: The number of rows in the set.
- **sym_update_user_data**: A user data area for keeping state across the symbol update chain. This area is allocated and initialized by the rtl_store_symbol_update_begin() function. This area is passed to each symbol update function call to maintain state. This area is de-allocated by the rtl_store_symbol_update_end() function.

**Return values**

If rtl_store_symbol_set_update() is successful, it returns RTL_SUCCESS.
If rtl_store_symbol_set_update() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_BAD_DATA</td>
<td>The storage system encountered a bad data error. For example, a column value does not match</td>
</tr>
<tr>
<td></td>
<td>the constraint for the column, or the data type of a column value is not valid for the column.</td>
</tr>
<tr>
<td></td>
<td>Check the symbol table definition and the constraints defined for the table to determine the cause</td>
</tr>
<tr>
<td></td>
<td>of the error.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are</td>
</tr>
<tr>
<td></td>
<td>broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store</td>
</tr>
<tr>
<td></td>
<td>remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_symbol_update_batch_begin()**

This function writes a batch of metadata to the store.

A symbol update can be broken down into several batches, as defined by the STOREXX_META_BATCH parameter. Defining batches helps to prevent possible problems with writing a large amount of data to a store within a single transaction. This function can be called multiple times during a symbol update; each call writes a batch of metadata to the store. This function is always called before any metadata is written, and after the rtl_store_symbol_update_begin() function is called.

**Syntax**

```c
int rtl_store_symbol_update_batch_begin(
    void *store,
    void *sym_update_user_data);
```

**Arguments**

- **store** The descriptor of the store to which metadata is being written. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **sym_update_user_data**
  A user data area for keeping state across the symbol update chain. This area is allocated and initialized by the rtl_store_symbol_update_begin() function. This area is passed to each symbol update function call to maintain state. This area is deallocated by the rtl_store_symbol_update_end() function.

**Return values**

If rtl_store_symbol_update_batch_begin() is successful, it returns RTL_SUCCESS.

If rtl_store_symbol_update_batch_begin() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_store_symbol_update_batch_end()**
This function is called at the end of a batch of symbol updates to a store.

A symbol update can be broken down into several batches, as defined by the STOREXX_META_BATCH parameter. Defining batches helps to prevent possible problems with writing a large amount of data to a store within a single transaction. This function is called after a batch of symbol updates is complete. It can be called multiple times during a symbol update (once for each batch). This function is called after the rtl_store_symbol_update_batch_begin function is called for the batch. No additional metadata is written until the rtl_store_symbol_update_batch_begin() function is called again.

If the function is called in commit mode, the store makes all of the changes for the batch permanent. If this function is called in commit mode and returns a failure, it is called again in rollback mode for the same batch.

If the function is called in rollback mode, the store rolls back the changes.

**Syntax**

```c
int rtl_store_symbol_update_batch_end(
    void *store,
    rtl_batch_end_mode_t mode,
    void *sym_update_user_data);
```

**Arguments**

- **store** The descriptor of the store to which metadata is written. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.
- **mode** The mode for ending the batch. If the batch is written with no errors, the store is asked to commit the batch. Otherwise, the store is asked to roll back the batch. The modes are:
  - **RTL_BATCH_COMMIT** Commits the changes in the batch to the store.
  - **RTL_BATCH_ROLLBACK** Rolls back the changes in the batch from the store.
- **sym_update_user_data** A user data area for keeping state across the symbol update chain. This area is allocated and initialized by the rtl_store_symbol_update_begin() function. This area is passed to each symbol update function call to maintain state. This area is deallocated by the rtl_store_symbol_update_end() function.

**Return values**

If rtl_store_symbol_update_batch_end() is successful, it returns RTL_SUCCESS.

If rtl_store_symbol_update_batch_end() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_store_symbol_update_begin()**

This function is called at the beginning of each symbol update.

This function must be called before any other symbol update function and after the rtl_store_open() function.

**Syntax**  
```c
int rtl_store_symbol_update_begin(
    void *store,
    void **sym_update_user_data);
```

**Arguments**

- **store**  The descriptor of the store to which metadata is written. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **sym_update_user_data**  A return pointer to a user data area for keeping state across the symbol update chain. This area is allocated and initialized by this function. This area is passed to each symbol update function call to maintain state. This area is deallocated by the rtl_store_symbol_update_end() function.

**Return values**

If rtl_store_symbol_update_begin() is successful, it returns RTL_SUCCESS.

If rtl_store_symbol_update_begin() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
**rtl_store_symbol_update_end()**
This function is called as part of the end of each symbol update.

This function is called once for each symbol update and is always called after the rtl_store_symbol_update_begin() function. Any symbol update functions that are called after this function is called must call the rtl_store_symbol_update_begin() function first.

**Syntax**
```c
int rtl_store_symbol_update_end(
    void *store,
    void *sym_update_user_data);
```

**Arguments**
- **store** The descriptor of the store to which metadata is written. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.
- **sym_update_user_data** A user data area for keeping state across the symbol update chain. This function deallocates this area and releases any other resources.

**Return values**
If rtl_store_symbol_update_end() is successful, it returns RTL_SUCCESS.

If rtl_store_symbol_update_end() fails, it returns an implementation-defined error code. DSE reports this error code in the message log.
Tick flushing interface

DSE uses this interface to flush ticks to the store.

The interface consists of the following functions:
- `rtl_store_discard_tick_set()`
- `rtl_store_flush_batch_begin()`
- `rtl_store_flush_batch_end()`
- `rtl_store_flush_begin()`
- `rtl_store_flush_end()`
- `rtl_store_flush_tick_set()`
**rtl_store_discard_tick_set()**

This function is called for every 100 ticks with bad data that is discarded to the discard table.

This function is called many times while the ticks are discarded to the discard table. For stores that use a simple schema, this function is called for each row set of simple schema ticks being discarded. This function is called after the `rtl_store_flush_batch_begin()` function and before the `rtl_store_flush_batch_end()` function.

**Syntax**

```c
rtl_store_discard_tick_set(
    void *store,
    void **discardtick_set,
    unsigned char **nulls_set,
    int num_fields,
    int *symbol_id_set,
    char **symbol_name_set,
    int set_size,
    void *flush_user_data);
```

**Arguments**

*store*   The descriptor of the store to which the data is discarded.

*discardtick_set*   An array of ticks with bad data to be discarded.

*nulls_set*   An array of null indicators for the ticks with bad data.

*num_fields*   The number of fields in the tick.

*symbol_id_set*   An array of symbol IDs for the ticks with bad data. This value is null for simple schema stores.

*symbol_name_set*   An array of symbol names for the ticks with bad data. This value is null for simple schema stores.

*set_size*   The number of ticks in the tick set to be discarded.

*flush_user_data*   A user data area for keeping state across the tick flush chain. This area is allocated and initialized by the `rtl_store_flush_begin()` function. This area is passed to each flush function call to maintain state. This area is de-allocated by the `rtl_store_flush_end()` function.

**Return values**

If `rtl_store_discard_tick_set()` is successful, it returns RTL_SUCCESS.
If rtl_store_discard_tick_set() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_BAD_DATA</td>
<td>The storage system encountered a bad data error.</td>
</tr>
<tr>
<td></td>
<td>For example, a column value does not match the constraint for the column, or the data type of a column value is not valid for the column. Check the discard table definition and the constraints defined for the table to determine the cause of the error.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_flush_batch_begin()**

This function is called before the flush process starts to flush ticks for a batch.

A flush can be broken down into several batches of a size determined by the configuration file. This avoids problems caused by flushing a large number of ticks at one time, like long transactions in a database.

This function can be called many times during a flush, once for each batch and is always called before any ticks are flushed for the symbol and after the rtl_store_flush_begin() function.

**Syntax**

```c
int rtl_store_flush_batch_begin(
    void *store,
    void *flush_user_data,
    rtl_store_rowset_mode_t rowset_mode);
```

**Arguments**

- `store` The descriptor of the store to which the ticks are flushed. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- `flush_user_data` A user data area for keeping state across the tick flush chain. This area is allocated and initialized by the rtl_store_flush_begin() function. This area is passed to each flush function call to maintain state. This area is deallocated by the rtl_store_flush_end() function.

- `rowset_mode` The mode in which a set of rows is inserted into the store. Valid values for this parameter are:

  - **RTL_STORE_ATOMIC_MODE**
    Atomic mode. In atomic mode, transactions are guaranteed either to be successfully processed or to be rolled back if an error occurs.

  - **RTL_STORE_NON_ATOMIC_MODE**
    Non-atomic mode. In non-atomic mode, the failure of part of the transaction does not cause the entire transaction to be rolled back.

**Return values**

If rtl_store_flush_batch_begin() is successful, it returns RTL_SUCCESS.

If rtl_store_flush_batch_begin() fails, it returns one of the following return codes:

**Table 29. Return codes**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
</tbody>
</table>
### Table 29. Return codes (continued)

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_flush_batch_end()**

This function is called after a flusher thread flushes ticks for a batch.

A flush can be broken into several batches of a size determined by the STOREXX_FLUSH_BATCH configuration parameter, to avoid problems caused by flushing a large number of ticks at a time (like long transactions in a database).

This function can be called many times during a flush, once for each batch, and it is always called after the rtl_store_flush_batch_begin() function for the batch. No more ticks will be flushed without calling the rtl_store_flush_batch_begin() function again. Depending on the mode, the store should either make all of the changes for that batch permanent (commit) or roll back the changes. If a call to the rtl_store_flush_batch_end() function to commit the batch returns a failure, the function is called again with a mode of rollback for the same batch.

**Syntax**

```
int rtl_store_flush_batch_end(
    void *store,
    rtl_batch_end_mode_t mode,
    void *flush_user_data);
```

**Arguments**

- **store** The descriptor of the store to which ticks are flushed. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **mode** The mode for ending the batch. If the batch proceeded with no errors, the store is asked to commit the batch. If an error occurred, the store is asked to roll back the batch. The modes are:
  - **RTL_BATCH_COMMIT**
    - To commit the batch to the store.
  - **RTL_BATCH_ROLLBACK**
    - To roll back the changes for the batch in the store.

- **flush_user_data** A user data area for keeping state across the tick flush chain. This area is allocated and initialized by the rtl_store_flush_begin() function. This area is passed to each flush function call to maintain state. This area is deallocated by the rtl_store_flush_end() function.

**Return values**

If rtl_store_flush_batch_end() is successful, it returns RTL_SUCCESS.

If rtl_store_flush_batch_end() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_DUPLICATE_KEY</td>
<td>An attempt to insert a tick or symbol in the store violates unique index rules. This return code is returned if high availability is enabled with shared databases. If high availability is not enabled, or it is enabled without shared databases, RTL_FAILURE is returned.</td>
</tr>
<tr>
<td>Return code</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried. This return code is returned if high availability is not enabled, or high availability is enabled without shared databases. If high availability is enabled with shared databases, RTL_DUPLICATE_KEY is returned.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_flush_begin()**
This function is called at the start of each flush of ticks to the store.

This function is called once for each flush and is always called before any other flush function and after the rtl_store_open() function.

**Syntax**
```c
int rtl_store_flush_begin(
    void *store,
    void **flush_user_data);
```

**Arguments**
- **store** The descriptor of the store to which ticks are flushed. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **flush_user_data** A return pointer to a user data area for keeping state across the tick flush chain. This area is allocated and initialized by this function. This area is passed to each flush function call to maintain state. This area is deallocated by the rtl_store_flush_end() function.

**Return values**
If rtl_store_flush_begin() is successful, it returns RTL_SUCCESS.

If rtl_store_flush_begin fails(), it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>- The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>- The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_flush_end()**

This function is called as part at the end of the flush of ticks to the store.

This function is called once for each flush and is always called after the rtl_store_flush_begin() function. The rtl_store_flush_begin() function must be called before any other flush functions can be called after this function.

**Syntax**

```c
int rtl_store_flush_end (  
    void *store,  
    void *flush_user_data);  
```

**Arguments**

- **store**  
  The descriptor of the store to which ticks are flushed. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **flush_user_data**  
  A user data area for keeping state across the tick flush chain. This function deallocates this area and releases any other resources.

**Return values**

If rtl_store_flush_end() is successful, it returns RTL_SUCCESS.

If rtl_store_flush_end() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
</tbody>
</table>
| RTL_FAILURE_RECONNECT      | The storage system encountered an error code that indicates one of the following conditions:  
                              - The connection to the data store is lost.  
                              - The connection still exists, but previously established resources for the connection are broken.  
                              Operations cannot proceed until the connection is restored. |
| RTL_FAILURE_RETRY          | The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried. |
**rtl_store_flush_tick_set()**

This function is called for each tick set being flushed to the store.

This function is called many times while the tick set is flushed, once for each row set that is discarded. For simple schema stores, this function is called for each set of simple schema ticks that are discarded. This function is called after the rtl_store_flush_batch_begin() function is called and before the rtl_store_flush_batch_end() function is called.

**Syntax**

```c
int rtl_store_flush_tick_set(
    void *store,
    void **tick_set,
    unsigned char **nulls_set,
    int num_fields,
    int *symbol_id_set,
    char **symbol_name_set,
    uint16 *status_set,
    int set_size,
    void *flush_user_data);
```

**Arguments**

- **store** The descriptor of the store to which the data is flushed.
- **tick_set** An array of ticks to be flushed.
- **nulls_set** An array of bitwise null indicators for the ticks to be flushed.
- **num_fields** The number of fields in the tick.
- **symbol_id_set** An array of symbol IDs for the ticks to be flushed. This value is null for simple schema stores.
- **symbol_name_set** An array of symbol names for the ticks to be flushed. This value is null for simple schema stores.
- **status_set** An array of status indicators for the unflushed data rows. A particular row returns:
  - 0 If the row status is successful (status_set[i] = 0).
  - Non-zero If the row status is not successful.
- **set_size** The number of rows in the tick set.
- **flusher_user_data** A user data area for keeping state across the tick flush chain. This area is allocated and initialized by the rtl_store_flush_begin() function. This area is passed to each flush function call to maintain state. This area is de-allocated by the rtl_store_flush_end() function.

**Return values**

If rtl_store_flush_tick_set() is successful, it returns RTL_SUCCESS.
If rtl_store_flush_tick_set() fails, it returns one of the following return codes:

*Table 33. Return codes*

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_BAD_DATA</td>
<td>The storage system encountered a bad data error. For example, a column value does not match the constraint for the column, or the data type of a column value is not valid for the column. Check the ticks table definition and the constraints defined for the table to determine the cause of the error.</td>
</tr>
</tbody>
</table>
| RTL_FAILURE_RECONNECT        | The storage system encountered an error code that indicates one of the following conditions:  
                                 • The connection to the data store is lost.  
                                 • The connection still exists, but previously established resources for the connection are broken.  

Operations cannot proceed until the connection is restored. |
| RTL_FAILURE_RETRY            | The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried. |
Tick scanning interface

DSE uses this interface to read ticks from the store

The interface consists of the following functions:

- \texttt{rtl\_store\_tick\_begin\_scan()}
- \texttt{rtl\_store\_tick\_end\_scan()}
- \texttt{rtl\_store\_tick\_get\_elem()}
- \texttt{rtl\_store\_tick\_get\_next()}
**rtl_store_tick_begin_scan()**

This function is called by the scanner threads before scanning the store for ticks.

This function is called once for each query of a store and is always called before the rtl_store_tick_get_next() function and after the rtl_store_open() function.

**Syntax**

```c
int rtl_store_tick_begin_scan(
    void   *store,
    int    *symbol_id,
    char   *symbol,
    rtl_datetime *start,
    rtl_datetime *end,
    int     flags,
    void    **scan_user_data);
```

**Arguments**

- **store**  The descriptor of the store which is being scanned for ticks. Can be used with descriptor accessor functions to retrieve information about the store.

- **symbol_id**  The ID of the symbol to have its ticks read from the store. If this is NULL then the symbol parameter in conjunction with the flags parameter determine the scan behavior.

- **symbol**  The name of the symbol or symbols (exact behavior dependent on the flags parameter) to have its ticks read from the store. This string is defined to be a NULL terminated string.

- **start**  The start time for the scan. If NULL then read from the earliest tick available.

- **end**  The end time for the scan. If NULL then read to the latest tick available.

- **flags**  The flags defining the desired behavior of the Scan. The flags parameter is a bit map with the following bit definitions:

  - **RTLSCAN_SKIP_START**  Look for data with timestamps greater than `start` instead of greater than or equal to `start`.

  - **RTLSCAN_SKIP_END**  Look for data with timestamps less than `end` instead of less than or equal to `end`.

  - **RTLSCAN_SET**  Look for data for multiple symbols; `name` is an SQL set expression.

- **scan_user_data**  A return pointer to a user data area for keeping state across the tick scan chain. This area is allocated and initialized by this function. This area is passed to each call to the rtl_store_tick_get_next() function to maintain state. This area is deallocated by the rtl_store_tick_end_scan() function.

**Return values**

If rtl_store_tick_begin_scan() is successful, it returns RTL_SUCCESS.
If rtl_store_tick_begin_scan() fails, it returns one of the following return codes:

Table 34. Return codes

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
</tbody>
</table>
| RTL_FAILURE_RECONNECT     | The storage system encountered an error code that indicates one of the following conditions:  
  • The connection to the data store is lost.  
  • The connection still exists, but previously established resources for the connection are broken.  
  Operations cannot proceed until the connection is restored. |
| RTL_FAILURE_RETRY         | The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried. |
**rtl_store_tick_end_scan()**

This function is called by the scanner threads after the store is scanned for all ticks matching the scan criteria.

It is called once for each query of a store and is always called after the rtl_store_tick_begin_scan() function. The rtl_store_tick_get_next() function will not be called after this function without calling the rtl_store_tick_begin_scan() function first.

**Syntax**

```c
int rtl_store_tick_end_scan(
    void *store,
    void *scan_user_data);
```

**Arguments**

- **store**  The descriptor of the store which is being scanned for ticks. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **scan_user_data**  A user data area for keeping state across the tick scan chain. This function deallocates this area and releases any other resources.

**Return values**

If rtl_store_tick_end_scan() is successful, it returns RTL_SUCCESS.

If rtl_store_tick_end_scan() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_tick_get_elem()**

This function is called by the scanner threads to search the store for a single tick which meets given time and symbol criteria.

This function returns a maximum of one tick. If more than one tick meets the criteria, this function returns the first qualifying tick that it finds. This function is called once for each query of a store and is always called after the rtl_store_open() function.

**Syntax**

```c
int rtl_store_tick_get_elem(
    void *store,
    int symbol_id,
    rtl_datetime *stamp,
    int cmp,
    void *tick,
    char *nulls,
    int num_fields);
```

**Arguments**

- **store** The descriptor of the store which is being scanned for ticks. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **symbol_id** The ID of the symbol to search for a tick element in the store.

- **stamp** The time for the element we are looking for in the store.

- **cmp** Defines the desired behavior for comparing the timestamp against the ticks to be retrieved. The `cmp` parameter has the following possible values:
  - RTL_LT - get data with a timestamp < "stamp"
  - RTL_LE - get data with a timestamp <= "stamp"
  - RTL_EQ - get data with a timestamp = "stamp"
  - RTL_GE - get data with a timestamp >= "stamp"
  - RTL_GT - get data with a timestamp > "stamp"

- **tick** The tick being returned.

- **nulls** A zero based array indicating which fields in the tick are Null in the tick being returned, which is populated by this function. A value of RTL_FIELD_IS_NULL indicates that the field is null, a value ofRTL_FIELD_IS_NOT_NULL indicates the field holds a valid non-null value.

- **num_fields** The number of fields in the tick.

**Return values**

If rtl_store_tick_get_elem() is successful, it returns RTL_SUCCESS. If there are no ticks to return, rtl_store_tick_get_elem() returns RTL_NO_MORE_RESULTS.
If `rtl_store_tick_get_elem()` fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken. Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
**rtl_store_tick_get_next()**

This function is called by the scanner thread when reading ticks from the store.

It is called repeatedly to retrieve all ticks meeting the criteria of the scan for that store. It is always called after the `rtl_store_tick_begin_scan()` function and before the `rtl_store_tick_end_scan()` function. It is called repeatedly until it indicates that there are no more ticks to return. No tick should be returned for the final invocation when no more tick is indicated by the return value.

**Syntax**

```c
int rtl_store_tick_get_next(
    void *store,
    void *tick,
    char *symbol_name,
    char *nulls,
    int num_fields,
    void *scan_user_data);
```

**Arguments**

- **store**: The descriptor of the store which is being scanned for ticks. Can be used in conjunction with descriptor accessor functions to retrieve information about the store.

- **tick**: The next tick being returned.

- **symbol_name**: The name of the symbol for the tick being returned by the store. This string is defined to be a NULL terminated string of length `MAX_SYMBOL_LEN` (including the NULL).

- **nulls**: A zero based array indicating which fields in the tick are Null in the tick being returned, which is populated by this function. A value of `RTL_FIELD_IS_NULL` indicates that the field is null, a value of `RTL_FIELD_IS_NOT_NULL` indicates the field holds a valid non-null value.

- **num_fields**: The number of fields in the tick.

- **scan_user_data**: A user data area for keeping state across the tick scan chain. This area is allocated and initialized by the `rtl_store_tick_begin_scan()` function. This area is passed to each call to the `rtl_store_tick_get_next()` function to maintain state. This area is deallocated by the `rtl_store_tick_end_scan()` function.

**Return values**

If `rtl_store_tick_get_next()` is successful, it returns `RTL_SUCCESS`. If there are no more ticks to return, `rtl_store_tick_get_next()` returns `RTL_NO_MORE_RESULTS`.

If `rtl_store_tick_get_next()` fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE</td>
<td>A fatal error occurred, or an error occurred for an operation that cannot be retried.</td>
</tr>
</tbody>
</table>

Chapter 25. DSE interfaces  241
<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_FAILURE_RECONNECT</td>
<td>The storage system encountered an error code that indicates one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• The connection to the data store is lost.</td>
</tr>
<tr>
<td></td>
<td>• The connection still exists, but previously established resources for the connection are broken.</td>
</tr>
<tr>
<td></td>
<td>Operations cannot proceed until the connection is restored.</td>
</tr>
<tr>
<td>RTL_FAILURE_RETRY</td>
<td>The storage system encountered an error code that indicates that the connection to the data store remains operational, and the failed operation can be retried.</td>
</tr>
</tbody>
</table>
High availability communication library interface

The high availability communication library interface defines the communication protocol between peer DSE instances in a high availability configuration.

This communication is needed to synchronize the peer DSE instances. DSE provides a default TCP/IP socket implementation.

The high availability communication library is a dynamically linked library that is loaded when high availability is enabled (when the HA_ENABLE configuration parameter is set to 1). This library is loaded before the feed handler library and the storage system library are loaded.

The high availability communication library interface functions are defined in rtl_gen_halib.h.

The interface consists of the following functions:

- rtl_ha_init()
- rtl_hacomm_connect()
- rtl_hacomm_disconnect()
- rtl_hacomm_listen()
- rtl_hacomm_peer_id()
- rtl_hacomm_receive()
- rtl_hacomm_send()
- rtl_ha_shutdown()
**rtl_ha_init()**

This function allocates and initializes the high availability user data area and a data structure for communicating with each of the peer DSE instances.

This function is called once after the high availability communication library is loaded. After the rtl_ha_init() function is called, the feed handler library API is loaded.

**Syntax**

```c
int rtl_ha_init(
    void **ha_user_data,
    int comm_buffer_size);
```

**Arguments**

- **ha_user_data (out)**
  A pointer to the high availability user data area.

- **comm_buffer_size (in)**
  The maximum size of messages sent.

**Return values**

If rtl_ha_init() is successful, it returns RTL_SUCCESS.

If rtl_ha_init() fails, it returns RTL_FAILURE.
rtl_hacomm_connect()

This function connects a standby DSE instance to the live DSE instance.

Syntax

```c
int rtl_hacomm_connect(
    void *ha_user_data,
    void **comm_link,
    char *remote_peer_id);
```

Arguments

- ha_user_data (in)
  A pointer to the high availability user data that is allocated by the rtl_ha_init() function.

- comm_link (out)
  A pointer to the communication control structure.

- remote_peer_id (out)
  The numeric identifier of the peer DSE instance in the configuration.

Return values

If rtl_hacomm_connect() is successful, it returns a pointer to the peer communication data structure and returns RTL_SUCCESS.

If rtl_hacomm_connect() fails, it returns RTL_FAILURE.
rtl_hacomm_disconnect()

This function closes a communication link. A DSE instance calls this function when it no longer needs this communication link or when it detects a communication error.

Syntax

```c
int rtl_hacomm_disconnect(void *comm_link);
```

Arguments

- **comm_link (in)**:
  A pointer to the communication control structure.

Return values

- If `rtl_hacomm_disconnect()` is successful, it returns RTL_SUCCESS.
- If `rtl_hacomm_disconnect()` fails, it returns RTL_FAILURE.
**rtl_hacomm_listen()**

This function enables the live DSE instance to listen for a connection from a peer DSE instance.

If the newly created communication link is from a peer DSE instance that has an existing communication link, the old communication link is closed.

**Syntax**

```c
int rtl_hacomm_listen(
    void *ha_user_data,
    void **comm_link);
```

**Arguments**

*ha_user_data* (in)

A pointer to the high availability user data area that is allocated by the `rtl_ha_init()` function.

*comm_link* (out)

A pointer to the communication control structure.

**Return values**

If `rtl_hacomm_listen()` is successful, it returns a pointer to the peer communication data structure and returns RTL_SUCCESS.

If `rtl_hacomm_listen()` fails, it returns RTL_FAILURE.
**rtl_hacomm_peer_id()**

This function returns the numeric identifier of the peer DSE instance in a specified communication link.

**Syntax**

```c
int rtl_hacomm_peer_id(void *comm_link);
```

**Arguments**

`comm_link` (in)

A pointer to the communication control structure.

**Return values**

If `rtl_hacomm_peer_id()` is successful, it returns a pointer to the peer communication data structure and returns RTL_SUCCESS.

If `rtl_hacomm_peer_id()` fails, it returns RTL_FAILURE.
**rtl_hacomm_receive()**

This function returns a pointer to the message buffer and its length, which are received on a specified communication link.

**Syntax**

```c
int rtl_hacomm_receive(
    void *comm_link,
    char **msg_buf,
    int *msg_len);
```

**Arguments**

- **comm_link (in)**
  A pointer to the communication control structure.

- **msg_buf (out)**
  A return pointer to buffer containing the message.

- **msg_len (out)**
  The number of bytes in the message.

**Return values**

If `rtl_hacomm_receive()` is successful, it returns RTL_SUCCESS.

If `rtl_hacomm_receive()` fails, it returns RTL_FAILURE.
**rtl_hacomm_send()**

This function sends a message on a specified communication link.

**Syntax**

```c
int rtl_hacomm_send(
    void *comm_link,
    char *msg_buf,
    int msg_len);
```

**Arguments**

**comm_link** (in)

A pointer to the communication control structure.

**msg_buf** (in)

A pointer to the message to send.

**msg_len** (in)

The number of bytes in the message.

**Return values**

If `rtl_hacomm_send()` is successful, it returns RTL_SUCCESS.

If `rtl_hacomm_send()` fails, it returns RTL_FAILURE.
**rtl_ha_shutdown()**

This function closes all high availability communications and prepares DSE for shutdown.

**Syntax**

```c
int rtl_ha_shutdown(void *ha_user_data);
```

**Arguments**

**ha_user_data (in)**

A pointer to the high availability user data area that is allocated by the `rtl_ha_init()` function.

**Return values**

If `rtl_ha_shutdown()` is successful, it returns RTL_SUCCESS.

If `rtl_ha_shutdown()` fails, it returns RTL_FAILURE.
Transport interface

The transport interface defines how DSE receives messages from feed data sources and transmits messages to feed data targets.

DSE provides pre-built implementations of this interface for files, TCP/IP sockets, and WebSphere MQ queues. You can use these supplied transports or you can implement your own.

The transport interface data structures, types, and function prototypes are defined in rtl_transport.h.

The interface consists of the following functions:
• rtl_transport_get_property()
• rtl_transport_handler_init()
• rtl_transport_handler_shutdown()
• rtl_transport_init()
• rtl_transport_shutdown()
• rtl_transport_source_batch_begin()
• rtl_transport_source_batch_end()
• rtl_transport_source_close()
• rtl_transport_source_delete_message()
• rtl_transport_source_get_message()
• rtl_transport_source_open()
• rtl_transport_source_put_message()
• rtl_transport_source_register_callback()
• rtl_transport_target_close()
• rtl_transport_target_open()
• rtl_transport_target_put_message()
**rtl_tpt_msgfmt_t data structure**

The `rtl_tpt_msgfmt_t` data structure holds information about the basic structure of source and target messages.

Messages are data packets that describe one or more events (for example, a stock tick). The `rtl_tpt_msgfmt_t` structure tells the transport how to extract a single message from the data source, and how to package a single message for an output data target. DSE calls a feed handler’s `rtl_feed_get_message()` function to initialize an `rtl_tpt_msgfmt_t` structure for both the data source and the data target. DSE then passes this information to the transport when the transport calls the `rtl_transport_init()` function.

**Syntax**

```c
typedef struct {
    rtl_tpt_msgtype msg_type;
    union {
        struct _fixedlen {
            int fixed_msg_length;
        } fixedlen;
        struct _varlen {
            int length_ind_start;
            int length_ind_length;
            rtl_tpt_lenfmt length_ind_fmt;
            int length_ind_incl;
        } varlen;
        struct _vardelim {
            unsigned char start_ind[MAX_IND_LEN];
            int start_ind_length;
            unsigned char end_ind[MAX_IND_LEN];
            int end_ind_length;
            unsigned char sep_ind[MAX_IND_LEN];
            int sep_ind_length;
        } vardelim;
    } u;
    int ok_to_truncate;
    int max_msg_length;
    void *user_data;
} rtl_tpt_msgfmt_t;
```

**Arguments**

- **msg_type**
  
  Defines the type of message to be processed by the transport. Valid values are:

  - **MSGTYPE_FIXED_LEN**
    
    Messages are all of an equal, fixed size, with no start or end indicators. The size of a message is specified by the `fixed_msg_length` field.

  - **MSGTYPE_VAR_LEN**
    
    Messages are variable length. The length of each message is determined by a length indicator embedded in the message itself. The fields that define this indicator are:

    - **length_ind_start**
      
      Defines the starting byte position of the length indicator. The first byte of a message is byte 0.

    - **length_ind_length**
      
      Defines the size of the length indicator itself. For example,
if the length indicator is a four-byte integer at the
beginning of a message, set length_ind_start = 0 and
length_end_length = 4.

**length_ind_fmt**
Defines the format of the length indicator. Valid values are:

- **LENFMT_ASCII_DECIMAL** specifies that the length
  indicator is an ASCII decimal number such as "123."
- **LENFMT_MSBYTE_FIRST** specifies that the length
  indicator is a binary number, with the most significant
  byte occurring first in the data stream. (This is the same
  as Network Byte Order.) Length indicators of this format
  can be 1, 2, or 4 bytes long.
- **LENFMT_LSBYTE_FIRST** specifies that the length
  indicator is a binary number, with the least significant
  byte occurring first in the data stream. Length indicators
  of this format can be 1, 2, or 4 bytes long.

**length_ind_incl**
Set this field to 1 if the value of the length indicator
includes the bytes of the length indicator itself, otherwise
set it to 0. For example, in a 50-byte message with a 4-byte
length indicator, set length_ind_incl to 0 if the length
indicator is byte 50, or set it to 1 if the length indicator is
byte 54.

**MSGTYPE_VAR_DELIM**
Messages are variable length. The beginning and end of each
message are demarcated by special start and end indicators. A
message must have at least an end indicator; the start indicator is
optional. The fields defining this indicator are:

**start_ind**
Defines the byte or bytes that make up the start indicator
and that specify the start of a message or group of
messages.

**start_ind_length**
Specifies the number of bytes in the start indicator. Set to 0
if the message does not include a start indicator.

**end_ind**
Defines the byte or bytes that make up the end indicator
and that specify the end of a message or group of
messages.

**end_ind_length**
Specifies the number of bytes in the end indicator. A
delimited message must always have an end indicator, so
this field cannot be set to 0.

**sep_ind**
In cases where multiple messages can appear between a
start indicator and an end indicator, sep_ind defines the
separator between the individual messages. This field does
not apply to data targets.
sep_ind_length
Specifies the number of bytes in the separator. Set to 0 if a separator is not needed. This field does not apply to data targets.

ok_to_truncate
This flag tells rtl_transport_source_get_message() how to deal with messages that are longer than max_msg_length. Set ok_to_truncate to 1 if you want the transport to return as much of the message as possible and throw away the remainder. Set ok_to_truncate to 0 if you want the transport to throw away the entire message and go on to the next message. This field does not apply to data targets.

max_msg_length
Specifies in bytes the maximum length of a message. This value may be used by a transport to size an internal buffer for holding a message.

user_data
A pointer to a user-defined data structure. This can be used by a custom feed handler to convey additional information to a custom transport. The pre-built transports do not attempt to interpret this information. The structure must be allocated and initialized by the rtl_feed_get_message_fmts() function.
**rtl_transport_get_property()**

This function enables DSE to determine various capabilities of a transport.

This function can be called multiple times during initialization.

**Syntax**

```c
int rtl_transport_get_property(
    rtl_tpt_prop prop,
    int *value);
```

**Arguments**

- `prop` (in)
  - The type of property DSE is inquiring about. This is an enumerated type; the possible values are:
    - `TPTPROP_DELIVERY_MODEL`
      - The data source delivery model of a transport. DSE recognizes two models:
        - **Push model**
          - DSE expects the source to pass a message to the transport when it has something to be processed. DSE establishes this delivery mechanism by calling the `rtl_transport_source_register_callback()` function.
        - **Pull model**
          - DSE initiates reading of messages from the data source by calling the `rtl_transport_source_get_message()` function.
      - If a transport does not use the push model, it must use the pull model.
    - `TPTPROP_SRC_SWITCH_TYPE`
      - Specifies the conditions under which DSE is allowed to switch to another data source. Data source switches are done by calling the `rtl_transport_source_close()` function on the current source, and calling the `rtl_transport_source_open()` function on the new source.
        - If a source switch can happen immediately, DSE performs the switch when the current source message has been fully processed.
        - If a source switch must be deferred, DSE waits until the `rtl_transport_source_get_message()` function returns `RTL_NO_MORE_MESSAGES` before performing the switch.
      - Sockets are an example of data sources for which immediate switches are typically acceptable. Files, on the other hand, are sources for which switches are generally postponed until the end of the first file is reached.
    - `TPTPROP_RESTARTABILITY`
      - Specifies what, if any, capability the data source has for supporting a restartable DSE configuration. Data sources can be one of the following types:
        - **Transactional**
          - For example, a WebSphere MQ queue. The data source supports a commit/rollback model, so that DSE restarts a load at the last committed read from the queue.
Non-restartable
For example, a socket. This data source is not supported in a restartable DSE configuration.

**TPTPROP_MSG_ID_LENGTH**
Specifies the length in bytes of the unique identifier of a message. This information is needed when DSE runs in restartable mode, to determine how much storage space to allocate each tick’s message ID.

value (out)
The value returned by this function for the requested property. The expected values for each property are:

**TPTPROP_DELIVERY_MODEL**
The transport delivery model. Possible values are:
- RTL_PUSH_MODEL
- RTL_PULL_MODEL

**TPTPROP_SRC_SWITCH_TYPE**
The source switch type. Possible values are:
- RTL_IMMEDIATE
- RTL_DEFERRED

**TPTPROP_RESTARTABILITY**
The capability of the data source to support a restartable DSE configuration. Possible values are:
- RTL_TRANSACTIONAL
- RTL_NONRESTARTABLE

**TPTPROP_MSG_ID_LENGTH**
The length in bytes of each message ID.
- For a file transport, the message ID length is either 4 or 8 (the size of an offset_t, in other words, the position within the data source file).
- For a WebSphere MQ transport, the message ID length is 24 (the size of the MsgId field in the message descriptor for a message).

**Return values**
If rtl_transport_get_property() is successful (the transport recognizes the requested property), it returns RTL_SUCCESS.

If rtl_transport_get_property() fails (the transport does not recognize the requested property), it returns RTL_FAILURE.
**rtl_transport_handler_init()**

This function is called by DSE to initialize a transport handler.

This function is called once for each transport handler, and is always called after rtl_transport_get_property() and before any other transport handler function.

**Syntax**

```c
int rtl_transport_handler_init(
    int transport_id,
    void **transport_handler_user_data);
```

**Arguments**

**transport_id (in)**

The ID of the transport being initialized (the XX in TRANSPORTXX_NAME, and so on).

**transport_handler_user_data (out)**

Returns a pointer to a user data area for storing information about the transport handler. This function allocates and initializes whatever user data it requires, and sets the transport_handler_user_data pointer to point to it. The user data is available to other functions in the transport, through the rtl_transport_get_user_data() function. User data is deallocated by the rtl_transport_handler_shutdown() function.

**Return values**

If rtl_transport_handler_init() is successful, it returns RTL_SUCCESS.

If rtl_transport_handler_init() fails, it returns RTL_FAILURE.
**rtl_transport_handler_shutdown()**

This function is called when a transport is shut down to close and free resources for a transport handler.

This function is called once, and is always called after the rtl_transport_handler_init() function and any other transport functions.

**Syntax**

```c
int rtl_transport_handler_shutdown(
    int transport_id,
    void *transport_handler_user_data);
```

**Arguments**

- **transport_id (in)**
  The ID of the transport that is being shut down.

- **transport_handler_user_data (in)**
  A user data area for storing information about the transport handler. This function deallocates this area and releases any other resources.

**Return values**

If rtl_transport_handler_shutdown() is successful, it returns RTL_SUCCESS.

If rtl_transport_handler_shutdown() fails, it returns RTL_FAILURE.
**rtl_transport_init()**

This function is called to initialize the transport for one feed.

This function is always called before any transport functions are called on the feed thread, and after the rtl_transport_handler_init() function is called. DSE can have multiple feeds configured. Each feed executes in its own thread, and each thread calls the rtl_transport_init() function on initialization.

**Syntax**

```c
int rtl_transport_init(
    int transport_id,
    rtl_tpt_msgfmt_t *source_msgfmt
    rtl_tpt_msgfmt_t *target_msgfmt
    void **transport_user_data);
```

**Arguments**

*transport_id* (in)
   The ID of the transport.

*source_msgfmt* (in)
   A pointer to a data structure that holds information about the basic structure of a source message. This structure is initialized by a prior call to rtl_feed_get_message_fmts().

*target_msgfmt* (in)
   A pointer to a data structure that holds information about the basic structure of a target message. This structure is initialized by a prior call to rtl_feed_get_message_fmts().

*transport_user_data* (out)
   A return pointer to a user data area for storing information about the transport for this feed. This function allocates and initializes whatever user data it requires, and sets the transport_user_data pointer to point to it. The user data is available to other functions in the transport. The user data is deallocated by the rtl_transport_shutdown() function.

**Return values**

If rtl_transport_init() is successful, it returns RTL_SUCCESS.

If rtl_transport_init() fails, it returns RTL_FAILURE.
**rtl_transport_shutdown()**

This function is called to shut down and free transport resources for a feed.

This function is always called after `rtl_transport_init()` and any other `rtl_transport_source` or `rtl_transport_target` functions. After `rtl_transport_shutdown()` is called, no `rtl_transport_source` or `rtl_transport_target` functions are called until `rtl_transport_init()` is invoked again. DSE can have multiple feeds; each feed executes in its own thread, and each thread calls `rtl_transport_shutdown()` on shutdown.

**Syntax**

```c
int rtl_transport_shutdown(
    int transport_id,
    void *transport_user_data);
```

**Arguments**

- **transport_id**
  The ID of the transport that is being shut down.

- **transport_user_data (in)**
  A user data area used for storing information about the transport for a feed. The `rtl_transport_shutdown()` function deallocates this area, and releases any other resources.

**Return values**

If `rtl_transport_shutdown()` is successful, it returns RTL_SUCCESS.

If `rtl_transport_shutdown()` fails, it returns RTL_FAILURE.
**rtl_transport_source_batch_begin()**

This function initiates a distributed transaction between the feed source and the storage system.

This function is called in conjunction with rtl_store_flush_batch_begin() in restartable mode. It is only called after rtl_transport_open() and cannot be called after rtl_transport_close(). It can be called many times during a flush, once for each batch and is always called before any rtl_transport_source_delete_message() calls.

**Syntax**

```c
int rtl_transport_source_batch_begin(
    int transport_id,
    void *transport_user_data);
```

**Arguments**

- **transport_id** (in)
  The ID of the transport.

- **transport_user_data** (in)
  A user data area for keeping state across the tick flush chain. This area is passed to each transport function call to maintain state.
  This area is allocated and initialized by the rtl_transport_init() function and is deallocated by the rtl_transport_shutdown() function.

**Return values**

If rtl_transport_source_batch_begin() is successful, it returns RTL_SUCCESS.

If rtl_transport_source_batch_begin() fails, it returns RTL_FAILURE.
**rtl_transport_source_batch_end()**

This function signals the end of a transaction between the feed source and the storage system, and either commits or rolls back the transaction.

This function is called in conjunction with rtl_store_flush_batch_end() in restartable mode. It can only be called after calling rtl_store_transport_batch_begin().

**Syntax**

```
int rtl_transport_source_batch_end(
    int transport_id,
    void *transport_user_data,
    rtl_batch_end_mode_t end_mode);
```

**Arguments**

- **transport_id (in)**
  The ID of the transport.

- **transport_user_data (in)**
  A user data area for keeping state across calls to the transport layer for the feed source. This area is passed to each transport function call.

  This area is allocated and initialized by the rtl_transport_init() function. It is deallocated by the rtl_transport_shutdown() function.

- **end_mode (in)**
  An enumerated type with the following values:

  - **RTL_BATCH_COMMIT**
    To commit a transaction.

  - **RTL_BATCH_ROLLBACK**
    To roll back or undo a transaction because an error occurred.

**Return values**

If rtl_transport_source_batch_end() is successful, it returns RTL_SUCCESS.

If rtl_transport_source_batch_end() fails to commit or roll back the transaction, it returns RTL_FAILURE.
**rtl_transport_source_close()**

This function is called to close a feed data source.

This function is always called after any transport source functions are invoked on the feed thread, and before the feed is shut down. A feed data source can be closed and opened many times during normal operation. The rtl_transport_source_close() function can be called repeatedly during a feed’s operation, but cannot be called again until rtl_transport_source_open() has been invoked.

**Syntax**

```c
int rtl_transport_source_close(
    int transport_id,
    void *transport_user_data);
```

**Arguments**

- `transport_id` (in)
  The ID of the transport.

- `transport_user_data` (in)
  The user data area allocated by rtl_transport_init() for storing information about the transport for this feed.

**Return values**

If rtl_transport_source_close() is successful, it returns RTL_SUCCESS.

If rtl_transport_source_close() fails, it returns RTL_FAILURE.
rtl_transport_source_delete_message()

This function deletes a message that was read (with a browse operation) and that remains accessible on the feed source.

When DSE is in restartable mode, this function is mainly used by the flusher thread to indicate that the ticks associated with it were written to the store. This function can also be called by a feed thread to delete messages that have no ticks.

Syntax

```c
int rtl_transport_source_delete_message (    int transport_id,    void *transport_user_data,    void *message_id);
```

Arguments

transport_id (in)
The ID of the transport.

transport_user_data (in)
A user data area for keeping state across calls to the transport layer for the feed source. This area is passed to each transport function call.

This area is allocated and initialized by the rtl_transport_init() function. It is deallocated by the rtl_transport_shutdown() function.

message_id (in)
The ID of the message to be deleted.

Return values

If rtl_transport_source_delete_message() succeeds, it returns RTL_SUCCESS.

If rtl_transport_source_delete_message() fails, it returns RTL_FAILURE.
**rtl_transport_source_get_message()**

This function is called to obtain one full message from the current data source.

This function is called after the data source is opened by the feed thread. This function can be called repeatedly during a feed’s normal operation, but cannot be called after rtl_transport_source_close() is invoked, unless rtl_transport_source_open() is called again.

A transport handler implementing the push model does not need to implement rtl_transport_source_get_message().

**Syntax**

```c
int rtl_transport_source_get_message(
    int transport_id,
    void *transport_user_data,
    unsigned char **msg_buf,
    int *msg_len,
    int *msgs_skipped
    int *bytes_read
    void *message_id);
```

**Arguments**

**transport_id** (in)

The ID of the transport.

**transport_user_data** (in)

A user data area allocated by rtl_transport_source_init() for storing information about the transport for this feed.

**msg_buf** (in/out)

A return pointer to a buffer containing a full message from the data source. Start and end delimiters, if present in the incoming data stream, are not included in this buffer. The transport is responsible for allocating and freeing this buffer. For better performance the buffer should be cached in the transport user data area and reused for each message.

**msg_len** (out)

The length, in bytes, of the message returned in msg_buf.

**msgs_skipped** (out)

The number of messages skipped by the transport before returning a message. This information is used for statistical purposes. Messages are skipped if they exceed max_msg_length and the ok_to_truncate flag is set to 0. These values are specified in the source message format descriptor.

**bytes_read** (out)

The number of bytes read from the data source. This information is used for statistical purposes.

**message_id** (out)

The ID of the message. This ID is used during flusher operation to mark the message as read.

**Return values**

If rtl_transport_source_get_message() is successful, it returns RTL_SUCCESS.

If rtl_transport_source_get_message() fails, it returns RTL_FAILURE.
If the data source is exhausted (for example, if an ASCII EOF is encountered in a file data source), rtl_transport_source_get_message() returns RTL_NO_MORE_MESSAGES.
**rtl_transport_source_open()**

This function is called to open a feed data source.

This function is always called after rtl_transport_init() is invoked, and before any other rtl_transport_source functions are called. A feed data source can be closed and opened many times during normal operation. The rtl_transport_source_open() function can be called repeatedly during a feed's operation, but cannot be called again until the rtl_transport_source_close() function is invoked.

**Syntax**

```c
int rtl_transport_source_open(
  int transport_id,
  void *transport_user_data,
  char *source);
```

**Arguments**

**transport_id (in)**
The ID of the transport.

**transport_user_data (in)**
The user data area allocated by rtl_transport_init() for storing information about the transport for this feed.

**source (in)**
A text string describing the data source to be opened. When a feed is first started this string is the value of the FEEDXX_SOURCE configuration parameter for this feed. When a feed source switch occurs, this string is the new_source argument from the rtlmode -fs command.

The format of the source name is defined by the transport. See "Source and target names for pre-built transports" on page 171 for information on the source name format for each type of pre-built transport.

**Return values**

If rtl_transport_source_open() is successful, it returns RTL_SUCCESS.

If rtl_transport_source_open() fails, it returns RTL_FAILURE.
**rtl_transport_source_put_message()**

This function is called to write a message to the feed data source.

This function is always called after the rtl_transport_source_open function on the feed thread. This function can be called repeatedly during a feed’s operation. It cannot be called after the rtl_transport_source_close function is called, until the rtl_transport_source_open function is called again.

**Syntax**

```c
int rtl_transport_source_put_message(
    int transport_id,
    void *transport_user_data,
    unsigned char *msg_buf,
    int msg_len,
    int *bytes_written);
```

**Arguments**

`transport_id (in)`

The ID of the transport.

`transport_user_data (in)`

The user data area allocated by rtl_transport_init() for storing information about the transport for this feed.

`msg_buf (in)`

The message buffer to be written to the feed data source. If length information or delimiters are specified in the source_msgfmt argument to rtl_transport_source_open(), the data that is written to the source is preceded and followed by the appropriate bytes.

`msg_len (in)`

The length of the message data in the message buffer.

`bytes_written (out)`

The number of bytes that are written to the source.

**Return values**

If this function is successful, it returns RTL_SUCCESS.

If this function fails, it returns RTL_FAILURE.
**rtl_transport_source_register_callback()**

This function is called to register a message callback function with a transport handler that implements a push model.

This function is always called after the feed data source is opened on the feed thread. The rtl_transport_source_register_callback() function can be called repeatedly during a feed’s normal operation, but cannot be called after rtl_transport_source_close() has been invoked, unless rtl_transport_source_open() is called again.

A transport handler that implements a pull model does not need to implement rtl_transport_source_register_callback().

**Syntax**

```c
int rtl_transport_source_register_callback(
    int transport_id,
    void *transport_user_data,
    void (*msg_cb_func)(unsigned char*, int, void*),
    void *cb_u_data);
```

**Arguments**

- **transport_id (in)**
  The ID of the transport.

- **transport_user_data (in)**
  The user data area allocated by rtl_transport_source_init() for storing information about the transport for this feed.

- **(msg_cb_func) (unsigned char*, int, void*) (in)**
  The DSE message callback function to be registered with the feed data source. The arguments for this message callback are a message buffer, a message length, and a user data area for storing DSE state and context information.

- **cb_u_data (in)**
  A pointer to the user data area for storing DSE state and context information. This user data area should be registered with the feed data source to be passed to the DSE message callback when invoked.

**Return values**

If rtl_transport_source_register_callback() is successful, it returns RTL_SUCCESS.

If rtl_transport_source_register_callback() fails, it returns RTL_FAILURE.
rtl_transport_target_close()

This function is called to close a feed data target.

This function is always called after any feed data target functions are called on the feed thread, and before the feed is shut down. A feed data target can be closed and opened many times during normal operation. This function can be called repeatedly during a feed’s operation, but cannot be called again until rtl_transport_target_open() is called.

Syntax

```c
int rtl_transport_target_close(
    int transport_id,
    void *transport_user_data);
```

Arguments

transport_id (in)

The ID of the transport.

transport_user_data (in)

The user data area allocated by rtl_transport_init() for storing information about the transport for this feed.

Return values

If rtl_transport_target_close() is successful, it returns RTL_SUCCESS.

If rtl_transport_target_close() fails, it returns RTL_FAILURE.
**rtl_transport_target_open()**

This function is called to open a feed data target.

This function is always called after rtl_transport_open() and before any rtl_transport_target functions. A feed data target can be closed and opened many times during normal operation. This function can be called repeatedly during a feed’s operation, but it cannot be called again until rtl_transport_target_close() is called.

**Syntax**

```c
int rtl_transport_target_open(
    int transport_id,
    void *transport_user_data,
    char *transport_target);
```

**Arguments**

**transport_id (in)**

The ID of the transport.

**transport_user_data (in)**

The user data area allocated by rtl_transport_init() for storing information about the transport for this feed.

**transport_target (in)**

A text string describing the data target to be opened. When the feed is first started this is the value of the FEEDXX_TARGET configuration parameter, where XX represents the feed ID. When a feed target switch occurs, the value of **transport_target (in)** becomes the new_target argument from the rtlmode -ft command line.

The format of the target name is defined by the transport. See [“Source and target names for pre-built transports” on page 171](#) for information on the target name format for each type of pre-built transport.

**Return values**

If rtl_transport_target_open() is successful, it returns RTL_SUCCESS.

If rtl_transport_target_open() fails, it returns RTL_FAILURE.
**rtl_transport_target_put_message()**

This function is called to write a message to the feed data target.

This function is always called after rtl_transport_target_open() on the feed thread. This function can be called repeatedly during a feed's normal operation, but cannot be called after rtl_transport_target_close() has been called unless rtl_transport_target_open() is invoked again.

**Syntax**

```c
int rtl_transport_target_put_message(
    int transport_id,
    void *transport_user_data,
    unsigned char *msg_buf,
    int msg_len);
```

**Arguments**

- **transport_id** (in)
  
  The ID of the transport.

- **transport_user_data** (in)
  
  The user data area allocated by rtl_transport_init() for storing information about the transport for this feed.

- **msg_buf** (in)
  
  The message buffer to be written to the feed data target. If delimiters and/or length information are specified in the target_msgfmt argument to rtl_transport_target_open(), the data written to the target will be preceded and followed by the appropriate bytes.

- **msg_len** (in)
  
  The length of the message data in the message buffer.

**Return values**

If rtl_transport_target_put_message() is successful, it returns RTL_SUCCESS.

If rtl_transport_target_put_message() fails, it returns RTL_FAILURE.
Chapter 26. Writing client applications

You can write C or Java client applications that connect directly to DSE to query and update data and retrieve statistics.

A client application can have multiple connections open either to a single instance of DSE or to multiple instances of DSE. A client application can query data using SQL.

DSE provides a client library, librtlcclient.a, for C client applications. The DSE wrapper also uses the librtlcclient.a library when connecting to DSE. DSE also provides the rtlapi.jar library for Java client applications.

This topic focuses on writing C client applications. The steps for writing Java client applications are similar. For more information about the Java API functions, see the Javadoc documentation that is located in the DSE_INSTALL_DIR/java/doc subdirectory (where DSE_INSTALL_DIR is the directory in which DSE is installed).

For C applications, the following API functions are available for managing connections to DSE:
• rtl_attach()
• rtl_detach()
• rtl_reattach()

For C applications, the following functions are available for retrieving data from data stores through DSE, getting statistics, and updating metadata:
• rtl_begin_scan()
• rtl_get_next()
• rtl_end_scan()
• rtl_elem()
• rtl_get_metadata()
• rtl_put_metadata()
• rtl_stats()
• rtl_api_version()
• rtl_get_tick_rec_info()
• rtl_get_meta_rec_info()

The following table shows the equivalent Java API functions for each of these C API functions.
### Table 38. C and Java API function equivalents

<table>
<thead>
<tr>
<th>C API function</th>
<th>Java equivalent</th>
<th>Java class</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl_api_version()</td>
<td></td>
<td>Version</td>
</tr>
<tr>
<td></td>
<td>RTL_MAJ_REV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTL_MIN_REV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTL_FP_REV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTL_BETA_REV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTL_Date_COMPILED</td>
<td></td>
</tr>
<tr>
<td>rtl_attach()</td>
<td>new</td>
<td>Connection</td>
</tr>
<tr>
<td>rtl_begin_scan()</td>
<td>beginScan</td>
<td>Connection</td>
</tr>
<tr>
<td>rtl_detach()</td>
<td>detach</td>
<td>Connection</td>
</tr>
<tr>
<td>rtl_elem()</td>
<td>getElem</td>
<td>Connection</td>
</tr>
<tr>
<td>rtl_end_scan()</td>
<td>endScan</td>
<td>RTLScan</td>
</tr>
<tr>
<td>rtl_get_metadata()</td>
<td>getMetaData</td>
<td>Connection</td>
</tr>
<tr>
<td>rtl_get_meta_rec_info()</td>
<td>getMetaFieldInfo</td>
<td>Connection</td>
</tr>
<tr>
<td>rtl_get_next()</td>
<td>getNext</td>
<td>RTLScan</td>
</tr>
<tr>
<td>rtl_get_tick_rec_info()</td>
<td>getTickFieldInfo</td>
<td>Connection</td>
</tr>
<tr>
<td>rtl_put_metadata()</td>
<td>putMetaData</td>
<td>Connection</td>
</tr>
<tr>
<td>rtl_reattach()</td>
<td>changeStore</td>
<td>Connection</td>
</tr>
</tbody>
</table>
Table 38. C and Java API function equivalents (continued)

<table>
<thead>
<tr>
<th>C API function</th>
<th>Java equivalent</th>
<th>Java class</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl_stats()</td>
<td>See the Javadoc for information about the specific functions.</td>
<td>DiscardStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DiscardStatDetail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FeedPartQStatDetail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FeedPartQsStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FeedStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FeedStatDetail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FlusherStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FlusherStatDetail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InsertsStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InsertStatDetail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LocksStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LockStatDetail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MetawriterStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MetawriterStatDetail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHMStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StoreStat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StoreStatDetail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SummaryStat</td>
</tr>
</tbody>
</table>

Related concepts

Chapter 21, “SQL queries with the DSE wrapper,” on page 137
You can query the data in DSE shared memory using SQL through the DSE wrapper. The wrapper enables you to create a nickname in the DB2 database system, which you can query using SQL to retrieve data from DSE. The DB2 database that is running the DSE wrapper can be on a remote system or on the same system as the DSE instance that is being queried.

Chapter 23, “Feed handlers,” on page 153
DSE uses feed handlers to receive and process data from different kinds of data feeds. The feed handler defines how the data is formatted and processed for the data feed.

Chapter 24, “Transports,” on page 163
The transport handles the feed communication transport. It simplifies the development effort required to create a feed handler by encapsulating the details of the feed transport mechanism.

Related reference

Chapter 25, “DSE interfaces,” on page 175
DSE defines interfaces that you must implement, for feed handlers, transports, storage systems, and high availability communications.
Connecting C applications to DSE

Every client application must connect to DSE before it can perform any other functions.

To connect to DSE, the C application must call the `rtl_attach()` function. This function returns descriptors that all of the other client application functions require in order to run.

Client applications can use the `rtl_attach()` function to connect to a store that uses a master-detail relationship. This function is not supported for stores that use a simple schema. However, regardless of the store schema type, the client application can still use the `rtl_attach()` function to query DSE statistics if the `storagesys_name` and `store_name` argument values are null.

Before the client application calls the `rtl_attach()` function, ensure that:
- DSE is running.
- The client application knows the name of the host system on which the DSE server is running.
- The configuration file contains all of the configuration parameter definitions that are listed in the following table. These definitions correspond to the arguments of the `rtl_attach()` function.

Table 39. `rtl_attach()` arguments and corresponding configuration parameters

<table>
<thead>
<tr>
<th>Configuration parameter</th>
<th><code>rtl_attach()</code> argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL_SERVICE_NAME</td>
<td>service_name</td>
</tr>
<tr>
<td>STORESYSXX_NAME</td>
<td>storagesys_name</td>
</tr>
<tr>
<td>STOREXX_NAME</td>
<td>store_name</td>
</tr>
<tr>
<td>GROUPXX_NAME</td>
<td>group_name</td>
</tr>
<tr>
<td>GROUPXX_PASSWORD</td>
<td>group_password</td>
</tr>
</tbody>
</table>

Connection information

The `rtl_attach()` function has a number of arguments to specify the system, store, and storage system to which you are connecting your application.

The following arguments indicate what you are connecting the application to:

**host_name**
Name of the machine where DB2 Data Stream Engine is running.

**service_name**
Name of the service from which DB2 Data Stream Engine receives requests.

**storagesys_name**
Name of the storage system that is defined in the rtlconfig file.

**store_name**
Name of the data store in the storage system.
For more information on these arguments, see Chapter 22, “Wrapper objects,” on page 141.

If you do not specify a store name or storage system name, your application can only call the rtl_get_stats() function.

**Connection authorization**

Each data store has an associated authorization level. To read data from a store, a user must have an authorization level that is greater than or equal to the authorization level for that store. A user’s authorization level is determined by the group to which the user belongs.

You must provide authentication information using the following arguments of the rtl_attach() function:

- **group_name**
  - Name of the group to which the user belongs.

- **group_password**
  - Unencrypted password for the group.

You can use the rtl_make_pass utility to generate encrypted passwords. See “rtl_make_pass utility” on page 49 for more information.

These arguments cannot have null values.

The following events occur when DSE authorizes a user to view a store:

1. The application attaching to DSE provides the client library with the group name, the group password in clear text form, and the name of the store that the user wants to access.
2. The DSE client library encrypts the password and passes it to DSE.
3. When DSE receives a connection request, it checks that the group name that the user sent is in the configuration file. If the group name is not found, the connection request is rejected.
4. DSE compares the encrypted password in the configuration file to the one sent by the user. If the passwords do not match, the connection request is rejected.
5. DSE compares the authorization level for the group in the configuration file to the authorization level of the store that is being opened. If the authorization level for the store is greater than the authorization level of the user’s group, the connection request is rejected.

DSE uses the following configuration parameters during the authorization procedure:

- GROUPXX_AUTHLEVEL
- GROUPXX_NAME
- GROUPXX_PASSWORD
- STOREXX_AUTHLEVEL

For a description of these parameters, see “Group configuration parameters” on page 57.
Connection failure due to version incompatibility

The rtl_attach() function can fail because of version incompatibility, as indicated by the RTLAPI_WRONG_VERSION return code. This means that the version of the DSE server does not match the version of the DSE client library.

To diagnose this problem, the client application should run the rtl_api_version() function. This function provides the version of the code of your application. Compare the API version number that the rtl_api_version() function provides with the DSE server version number that is in the server log file.

Reconnecting C applications

Use the rtl_reattach() function to reconnect your C application to an existing data store or connect it to a new data store.

Call the rtl_reattach() function in the following situations if:

- Your C application is getting information from a data store, and you want to get information from a different data store.
- You get one of the following error conditions:
  - RTLAPI_ATTACH_REFUSED (if the store uses a simple schema)
  - RTLAPI_IO_FAILED
  - RTLAPI_NOT_CONNECTED

This function is not supported for stores that use a simple schema. However, regardless of the store schema type, the client application can still use the rtl_reattach() function to query DSE statistics if the storage_sys_name and store_name argument values are null.

Querying data in DSE data stores

A client application can call a number of API functions to retrieve data from a DSE data store.

When you run a query for data in a DSE store, DSE first checks shared memory for the data. If the data is not in memory and DSE determines that the data is on disk, it checks for the data on disk using the appropriate storage system. DSE automatically determines if it needs to retrieve the data from shared memory, the storage system, or both, to satisfy the query.

Queries perform best when they search for data that is entirely in shared memory. For example, if shared memory holds the data since 9:00 a.m., queries that specify a date and time range at or after 9:00 a.m. run the most efficiently. If you specify a date and time range earlier than 9:00 a.m., data must be read from the storage system.

**Prerequisite:** Before attempting to retrieve data from a DSE store, ensure that your application called the rtl_attach() function to establish the connection to DSE. If you are connected to DSE but the call to rtl_attach() did not connect to the required store, you can use the rtl_reattach() function to connect the application to the required store.

Querying a range of data

A client application can query a range of data from DSE by using a scan.
The client application starts the scan by calling the `rtl_begin_scan()` function. The scan requests data for a specified symbol between a starting and ending timestamp. The symbol can be a single symbol, a set of symbols, or null to indicate all symbols.

After the scan is started, the result records can be retrieved by calling the `rtl_get_next()` function repeatedly until it returns the RTL_NO_MORE_RESULTS return code. Each result record is returned with each successive call to the `rtl_get_next()` function. The client application cannot return to previous results.

After all of the required results are processed, the client application can end the scan with the `rtl_end_scan()` function. After starting a scan, you can end it after retrieving none, some, or all of the result records.

Restrictions:
- You cannot start a new scan before ending a previous scan.
- No other requests can be sent to DSE when a scan is in progress.

The tick records are returned as a data buffer and an array of null indicators. See “Interpreting result records returned by DSE” on page 282 for information on how to interpret these results.

The following code sample shows a scan for all tick records for the symbol IBM, starting at the time 2001/02/04 14:32:24.00013:

```c
strcpy(sym,"IBM");
start = 2001020414322400013LL;
flags = 0;
err = 0;
Cnt = 0;

rscan = (rtlscan *) rtl_begin_scan(rdesc, sym, &start, 0, flags, &err);
if (err != RTLAPI_SUCCESS) {
    printf("rtl_begin_scan failed, error = %d\\n", err);
    exit(-1);
}

while ((ret=rtl_get_next(rscan, (void **)&tick, nulls, &name))
    != RTLAPI_NO_MORE_RESULTS) {
    if (ret != RTLAPI_SUCCESS) {
        printf("rtl_get_next failed, error = %d\\n", ret);
        break;
    }
    printf("Tick record found: %s\\n", name);
   ighbor = tick;
    Cnt++;
}

rtl_end_scan(rscan);
printf("Scan complete. Got %d ticks\\n", Cnt);
```

**Querying a single element at a point in time**

A client application can query a specific tick record from the DSE data store by calling the `rtl_elem()` function.

This function retrieves a record for a specified symbol, satisfying a given comparison operator for a given timestamp.

The symbol can be a single symbol name only.

The comparison operator can be one of the following:
<= Less than or equal to
< Less than
= Equal to
> Greater than
>= Greater than or equal to

The rtl_elem() function returns the first record for the specified symbol with a timestamp that satisfies the given comparison operator for the given timestamp.

The tick record is returned as a data buffer and an array of null indicators. See [Interpreting result records returned by DSE](#) for information on how to interpret these results.

**Querying metadata**

A client application can query the metadata record for a given symbol from the DSE data store by calling the rtl_get_metadata() function.

The metadata record is returned as a data buffer and an array of null indicators. See [Interpreting result records returned by DSE](#) for information on how to interpret these results.

**Interpreting result records returned by DSE**

A client application can get information about the structure of the tick and metadata records for the DSE data store by calling the rtl_get_tick_rec_info() and rtl_get_metadata_rec_info() functions.

These functions return the following information:

- Record size
- Number of fields
- An array of field information

The field information includes the type, offset, size, default value, and name. See the rtl_rectype.h header file for the content of this structure. The tick and metadata record structure information is based on the schema that is defined in the DSE configuration file for the record type that is used by the DSE data store being queried. The schema is defined by the RECORDXX_SCHEMA and RECORDXX_META_SCHEMA configuration parameters.

For more information, see:

- [“Record configuration parameters” on page 61](#)
- Chapter 10, “Supported data types,” on page 83

**Updating symbol metadata in a data store**

A client application can update the metadata for a given symbol in a DSE data store by calling the rtl_metadata_set() function.

The metadata can be updated one field at a time. The client application must indicate the field to change and provide the new value to be assigned to the field.

**Prerequisite:** Before attempting to update metadata in a store, ensure that your client application called the rtl_attach() function to establish the connection to DSE.
If you are connected to DSE but the call to `rtl_attach()` did not connect to the required store, you can use the `rtl_reattach()` function to connect the application to the required store.

---

**Querying statistics from DSE**

A client application can query DSE statistics by calling the `rtl_stats()` function. DSE provides several different statistics about different areas of its operations, such as shared memory, new symbol insertion, and feeds.

Before attempting to query statistics, ensure that your client application called the `rtl_attach()` function to establish the connection to DSE. The connection does not need to be attached to any specific store to query statistics.

The client application indicates which statistic it requires and the result is the appropriate statistic structure, as defined in the `rtl_client.h` header file.

The following code sample shows a function to display the shared memory statistics returned by DSE:

```c
void print_shm_stats(stat_response *stat_resp)
{
    rtl_stat_shm *ss = &stat_resp->rtlstat_u.rtlstat_shm;

    printf("Shm Statistics:
" );
    printf("Shm size = %lld\n", ss->rtlstat_shm_size);
    printf("SecHdr used = %lld\n", ss->rtlstat_sechdr_used);
    printf("Total ticks = %lld\n", ss->rtlstat_tot_ticks);
    printf("Ticks used = %lld\n", ss->rtlstat_ticks_used);
    printf("Ticks loaded = %lld\n", ss->rtlstat_ticks_loaded);
    printf("Ticks purged = %lld\n", ss->rtlstat_ticks_purged);
    printf("Total sechdrs = %d\n", ss->rtlstat_tot_sechdrs);
}
```

---

**Disconnecting C applications from DSE**

To disconnect a C application from the data store and from DSE, call the `rtl_detach()` function.

You should always disconnect an application when it is not used, to free up resources and improve performance. A query thread between the application and DSE remains active until you call the `rtl_detach()` function, or until the application terminates.

If you receive an error and calling the `rtl_reattach()` function does not fix the problem, try the following actions:

1. Disconnect the application by calling the `rtl_detach()` function.
2. Call the `rtl_attach()` function to establish a new connection.

If your application crashes before you disconnect the application, DSE disconnects the application automatically. You need to run `rtl_attach()` again to establish a connection.
Chapter 27. DSE API reference

DSE provides a number of application programming interface (API) functions.

These functions enable you to:
- Write client applications that communicate with DSE.
- Implement the interfaces for storage systems, feed handlers, and high availability communications.

Related concepts
Chapter 21, “SQL queries with the DSE wrapper,” on page 137
You can query the data in DSE shared memory using SQL through the DSE wrapper. The wrapper enables you to create a nickname in the DB2 database system, which you can query using SQL to retrieve data from DSE. The DB2 database that is running the DSE wrapper can be on a remote system or on the same system as the DSE instance that is being queried.

Related tasks
Chapter 26, “Writing client applications,” on page 275
You can write C or Java client applications that connect directly to DSE to query and update data and retrieve statistics.

Types of API functions

Some of the API functions provided by DSE are specific types of functions.

All of the API functions are described in “DSE API functions” on page 293. However, not all of these functions are classified in these specific types.

The types of API functions are:
- C client API
- Feed handler API
- High availability API:
  - High availability state API
  - High availability accessor API
- Java client API
- Record type API
- Store API
- Storage system API
- Time API
- Transport API

C client API

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.

Use the following API functions to manage connections to DSE:
- rtl_attach()
Use the following API functions to retrieve data from data stores through DSE:

- rtl_detach()
- rtl_reattach()

- rtl_api_version()
- rtl_begin_scan()
- rtl_elem()
- rtl_end_scan()
- rtl_get_metadata()
- rtl_get_meta_rec_info()
- rtl_get_next()
- rtl_get_tick_rec_info()
- rtl_put_metadata()
- rtl_stats()

**Related reference**

"rtl_attach()” on page 296
This function allows a user to attach to DSE. You must call the rtl_attach() function before any data can be transferred from DSE to a client.

"rtl_detach()” on page 303
This function disconnects a C client application from the data store and from DSE. This function detaches the connection that was established by the rtl_attach() function.

"rtl_reattach()” on page 354
This function switches to a new data store once a connection has been established by the rtl_attach() function.

"rtl_api_version()” on page 295
This function retrieves the API version of the C client API Library.

"rtl_begin_scan()” on page 298
To retrieve a range of data from DSE, you must first start a scan. The scan descriptor determines what data will be returned. The rtl_begin_scan() function allows you to select all or a subset of the symbols in the store and a time range.

"rtl_elem()” on page 304
This function retrieves a tick record for a point in time from the data store.

"rtl_end_scan()” on page 306
This function ends a scan for tick data that was started with the rtl_begin_scan() function.

"rtl_get_metadata()” on page 324
This function reads metadata from shared memory for a specified symbol. This function returns the entire metadata structure for the specified symbol.

"rtl_get_meta_rec_info()” on page 327
This function retrieves information about the metadata structure that is associated with the rtldesc that you specify.

"rtl_get_next()” on page 328
This function retrieves the next tick record for a scan query.

"rtl_get_tick_rec_info()” on page 335
This function retrieves information about the tick data structure that is associated with the rtldesc that you specify.

"rtl_put_metadata()” on page 352
This function updates one field of a symbol’s metadata in shared memory.
Feed handler API

DSE provides API functions for feed handlers to retrieve information about the feed.

Related concepts

- "Feed handler reference and context information" on page 156
  DSE provides API functions that enable feed handlers to retrieve information about the DSE configuration.

Related reference

- "rtl_feed_get_gapfill()" on page 307
  This function returns the gapfill for the feed.
- "rtl_feed_get_handler_id()" on page 308
  This function returns the handler ID for the feed.
- "rtl_feed_get_handler_name()" on page 309
  This function returns the handler name for the feed.
- "rtl_feed_get_info()" on page 310
  This function returns the feed information for the feed.
- "rtl_feed_get_source()" on page 311
  This function returns the current source for the feed.
- "rtl_feed_get_store_id()" on page 312
  This function returns the store ID for the feed.
- "rtl_feed_get_store_name()" on page 313
  This function returns the store name for the feed.
- "rtl_feed_get_target()" on page 314
  This function returns the current target for the feed.
- "rtl_feed_get_transport_id()" on page 315
  The rtl_feed_get_transport_id() function returns a transport ID for the corresponding feed ID.
- "rtl_feed_get_user_data()" on page 316
  This function returns the feed user data for the feed. The feed user data is the memory buffer allocated by the feed handler and is returned to DSE from the rtl_feed_init() call in the feed handler implementation. The contents and use of that area are determined by the feed handler implementation.
- "rtl_feed_is_configured()" on page 317
  This function enables you to determine if a particular feed is defined in the configuration file. You can use this function to check the validity of a feed ID before calling other feed accessor functions, such as the rtl_feed_get_info() function.
- "rtl_handler_get_id_from_name()" on page 342
  This function returns the handler ID for the feed handler.
- "rtl_handler_get_info()" on page 343
  This function returns the handler information for the feed handler.
- "rtl_handler_get_name()" on page 344
  This function returns the name for the feed handler.
- "rtl_handler_get_user_data()" on page 345
  This function returns the handler user data for the feed handler. The handler user data is the memory buffer allocated by the feed handler and is returned to DSE from the rtl_handler_init() call in the feed handler implementation.
This function enables you to determine if a particular feed handler is defined in the configuration file. You can use this function to check the validity of a feed handler ID before calling other handler accessor functions, such as rtl_handler_get_info().

High availability API
DSE has interfaces to support high availability configurations. The high availability communication library API enables communication between peer DSE instances in a cluster of DSE instances. The high availability control API provides functions to manage the high availability states of DSE instances.

The high availability API functions can be further classified as high availability state APIs and high availability information accessor APIs.

High availability state API
The high availability state API provides functions that custom code can use in a DSE instance to manage the high availability state of DSE.

The following code defines the values for each of the high availability states. This code is included in the rtl_state.h file:

```c
typedef enum {
    RTL_STATE_ACTIVE = 0,
    RTL_STATE_STARTED = 1,
    RTL_STATE_SYNCHRONIZING = 2,
    RTL_STATE_STANDBY = 3,
    RTL_STATE_INACTIVE = 4
} rtl_state_t;
```

Related reference
- “rtl_state_change_register()” on page 368
  This function registers a function to be called whenever the state of the DSE instance changes.
- “rtl_state_get()” on page 369
  This function returns the current state of the DSE instance. This function performs the same task as the rtlstat -h command.
- “rtl_state_get_info()” on page 370
  This function returns the user information that is supplied in the HA_INFO configuration parameter.
- “rtl_state_set()” on page 371
  This function changes the state of a DSE instance.

High availability information accessor API
DSE provides API functions for the high availability communication library to use. The purpose of these API functions is to retrieve information about the high availability configuration.

Related reference
- “rtl_ha_get_peer_id_from_name()” on page 338
  This function returns a numeric ID value for the specified peer DSE instance.
- “rtl_ha_get_peer_info_from_id()” on page 339
  This function returns the information about the peer instance that corresponds to the specified peer ID.
- “rtl_ha_get_peer_name_from_id()” on page 340
  This function returns the name of the peer instance that corresponds to the specified peer ID.
The rtl_ha_log function enables you to display messages in the DSE message log. The message file that a high availability library uses is defined by the HA_MSG_PATH parameter in the configuration file.

This function returns the user information that is supplied in the HA_INFO configuration parameter.

This function returns a pointer to the high availability user data area that is allocated by the rtl_ha_init() function.

Java client API
For information about the Java API functions, see the Javadoc documentation that is located in the DSE_INSTALL_DIR/java/doc subdirectory (where DSE_INSTALL_DIR is the directory in which DSE is installed).

Record type API
DSE provides API functions for retrieving information about record types and setting default values for record types.

Related reference

This function gets information about the metadata for a record type.

This function returns the record type ID for the record.

This function gets information about the tick for a record type.

This function sets a provided metadata area and corresponding nulls array to the default values specified by the RECORDXX_META_SCHEMA parameter in the configuration file.

This function sets a provided tick and corresponding nulls array to the default values specified by the RECORDXX_SCHEMA parameter in the configuration file.

This function converts a bitwise array to a byte array.

Store API
DSE provides this API for accessing information about a store from either the store descriptor or a store ID.

Related reference

The store interface supports a generic store of ticks and symbols. DSE uses this interface to open and close a store.

This function returns the tick flush batch size from the store descriptor.

This function returns the tick flush batch size from the store ID.

This function returns the store ID from the store descriptor.
Storage system API

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.

Related reference

“rtl_storesys_get_conn_from_desc()” on page 412
This function returns the storage system conn from the storage system descriptor.

“rtl_storesys_get_conn_from_id()” on page 413
This function returns the storage system conn from the storage system ID.

“rtl_storesys_get_ha_from_id()” on page 414
This function returns the value of the STORESYSXX_HA parameter in the configuration file. This parameter specifies whether a storage system is configured for high availability support.
**rtl_storesys_get_id_from_desc()** on page 415
This function returns the storage system ID from the storage system descriptor.

**rtl_storesys_get_id_from_name()** on page 416
This function returns the ID of the given storage system name.

**rtl_storesys_get_info_from_desc()** on page 417
This function returns the storage system info from the storage system descriptor.

**rtl_storesys_get_info_from_id()** on page 418
This function returns the storage system info from the storage system ID.

**rtl_storesys_get_name_from_desc()** on page 419
This function returns the storage system name from the storage system descriptor.

**rtl_storesys_get_name_from_id()** on page 420
This function returns the name of the storage system with the given ID.

**rtl_storesys_get_password_from_desc()** on page 421
This function returns the unencrypted storage system password from the storage system descriptor.

**rtl_storesys_get_password_from_id()** on page 422
This function returns the unencrypted storage system password from the storage system ID.

**rtl_storesys_get_user_from_desc()** on page 423
This function returns the storage system user from the storage system descriptor.

**rtl_storesys_get_user_from_id()** on page 424
This function returns the storage system user from the storage system ID.

**rtl_storesys_is_configured()** on page 425
This function enables you to determine if a particular storage system is defined in the configuration file. You can use this function to check the validity of a storage system ID before calling other storage system accessor functions, such as the rtl_storesys_get_info() function.

**rtl_storesys_log()** on page 426
This function provides a means for messages to be displayed in DSE message log.

## Time API

DSE provides API functions for managing timestamps.

**Related reference**

**rtl_add_seconds()** on page 294
This function adds a number of seconds to an rtl_datetime type timestamp.

**rtl_get_rtltime_date_fields()** on page 331
This function gets the date components of an rtl_datetime type.

**rtl_get_rtltime_fields()** on page 332
This function gets the date and time components of an rtl_datetime type.

**rtl_get_rtltime_time_fields()** on page 333
This function gets the time components of an rtl_datetime type.

**rtl_gmtime_to_rtltime()** on page 337
This function generates an rtl_datetime type timestamp for the current GMT time.

**rtl_localtime_to_rtltime()** on page 350
This function generates an rtl_datetime type timestamp for the current local time.
"rtl_minus_seconds()" on page 351
This function subtracts a number of seconds from an rtl_datetime type timestamp.

"rtl rtljavadate_to_rtltime()" on page 356
This function converts a timestamp from an rtl_java_date type to an rtl_datetime type.

"rtl_rtltime_cmp()" on page 357
This function compares two rtl_datetime timestamps.

"rtl_rtltime_cpy()" on page 358
This function copies a source rtl_datetime timestamp to a target rtl_datetime timestamp.

"rtl_rtltime_date_to_string()" on page 359
This function converts the date part of an rtl_datetime type timestamp to a string representation in ISO 8601 format YYYY-MM-DD.

"rtl_rtltime_time_to_string()" on page 360
This function converts the time part of an rtl_datetime type timestamp to a string representation in ISO 8601 format HH:MM:SS.

"rtl_rtltime_to_rtljavadate()" on page 361
This function converts a timestamp from an rtl_datetime type to an rtl_java_date type.

"rtl_rtltime_to_string()" on page 362
This function converts a timestamp from an rtl_datetime type to a timestamp string representation in ISO 8601 format YYYY-MM-DD HH:MM:SS:FFFF.

"rtl_rtltime_to_timet()" on page 363
This function converts a timestamp from an rtl_datetime type to a time_t type.

"rtl_rtltime_to_tm()" on page 364
This function converts a timestamp from an rtl_datetime type to a tm struct type.

"rtl_set_rtltime_fields()" on page 366
This function sets the date and time components of an rtl_datetime type.

"rtl_timet_to_rtltime()" on page 427
This function converts a timestamp from a time_t type to an rtl_datetime type.

"rtl_tm_to_rtltime()" on page 428
This function converts a timestamp from a tm struct type to an rtl_datetime type.

**Transport API**

DSE provides this API for transport handlers to retrieve information about the transport.

Related reference

"rtl_transport_get_info()" on page 429
This function returns the value of the TRANSPORTXX_INFO configuration parameter.

"rtl_transport_get_name()" on page 430
This function returns the value of the TRANSPORTXX_NAME configuration parameter, where XX is the ID of the specified transport.

"rtl_transport_get_user_data()" on page 431
This function returns a transport handler's user data. The transport user data is a memory buffer allocated by the transport, and is returned to DSE by rtl_transport_handler_init(). The contents and use of this buffer are determined by the transport implementation.
“rtl_transport_is_configured()” on page 432
This function enables you to determine if a particular transport is defined in the configuration file. Use this function to check the validity of a transport ID before calling other transport accessor functions such as rtl_transport_get_info().

“rtl_transport_log()” on page 433
This function allows functions to write messages to the log file.

**DSE API functions**

Use these API functions to write client applications that communicate with DSE and to implement the interfaces for storage systems, feed handlers, and high availability communications.

These API functions are valid in one or more of the following environments:
- Client application
- Storage system
- Feed handler
- High availability library
- Transport

The information in these topics indicates where each API function is valid.
**rtl_add_seconds()**

This function adds a number of seconds to an rtl_datetime type timestamp.

The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
void rtl_add_seconds(
    rtl_datetime *rtl_time,
    long       inc,
    rtl_datetime *ret);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_time**
  
  The original timestamp as an rtl_datetime type (input parameter).

- **inc**
  
  The number of seconds to add to the original timestamp (input parameter).

- **ret**
  
  The resulting timestamp with added seconds as an rtl_datetime type.

**Return values**

The rtl_add_seconds() function has no return values.

**Related concepts**

- "Time API" on page 291
  
  DSE provides API functions for managing timestamps.
rtl_api_version()  
This function retrieves the API version of the C client API Library.

**Syntax**
```c
void rtl_api_version(  
    int *major_ver,  
    int *minor_ver,  
    int *fixpack_ver,  
    int *beta_ver,  
    char **date)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **major_ver**
  - Retrieves the major version number of the API library.

- **minor_ver**
  - Retrieves the minor version number of the API library.

- **fixpack_ver**
  - Retrieves the fixpack version number of the API library.

- **beta_ver**
  - Retrieves the beta version number if any of the API library.

- **date**
  - Points to the date that the API library was compiled.

**Return values**

The rtl_api_version() function returns the values that are specified by the arguments of the function.

**Related concepts**

"C client API” on page 285
DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_attach()**

This function allows a user to attach to DSE. You must call the rtl_attach() function before any data can be transferred from DSE to a client.

Client applications can use the rtl_attach() function to get statistics for stores that use a simple schema, if the **storagesys_name** and **store_name** argument values are null.

**Syntax**

```c
int rtl_attach(char *host_name, 
                char *service_name, 
                char *storagesys_name, 
                char *store_name, 
                char *group_name, 
                char *group_password, 
                int *err)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

**host_name**

Name of the machine where DSE is running.

If **host_name** is null, the function uses the environment variable RTL_HOSTNAME. If both **host_name** and RTL_HOSTNAME are null, you will receive an error.

**service_name**

Name of the service from which DSE receives requests.

If **service_name** is null, the function uses the environment variable RTL_SERVICENAME. If both **service_name** and RTL_SERVICENAME are null, you will receive an error. If **service_name** contains all digits, DSE assumes that it is a port number. If **service_name** contains any alphabetic characters, DSE assumes that it is a service name.

**storagesys_name**

Name of the storage system that is defined in the **rtlconfig** file. **storagesys_name** can be null. This value is null for stores that use a simple schema.

**store_name**

Name of the data store in the storage system. **store_name** can be null. This value is null for stores that use a simple schema.

**group_name**

Name of the group to which the user belongs. **group_name** cannot be null.

**group_password**

Unencrypted password for the group. **group_password** cannot be null.
err  Return code status.

Return values

If rtl_attach() is successful, it sets the err argument to RTLAPI_SUCCESS.

If rtl_attach() fails, it returns null and sets the err argument to one of the following return codes:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_ATTACH_REFUSED</td>
<td>This function does not support stores that use a simple schema.</td>
</tr>
<tr>
<td>RTLAPI_BAD_HOSTNAME</td>
<td>host_name is null, longer than 32 characters, or could not be reached.</td>
</tr>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Connection to DSE failed or dropped.</td>
</tr>
<tr>
<td>RTLAPI_NOT_AUTH</td>
<td>Authority of the store is greater than the authority of the group.</td>
</tr>
<tr>
<td>RTLAPI_NO_SUCHSTORESYS</td>
<td>Storage system does not exist.</td>
</tr>
<tr>
<td>RTLAPI_NO_SUCHSTORE</td>
<td>Store name does not exist in the storage system space.</td>
</tr>
<tr>
<td>RTLAPI_WRONG_VERSION</td>
<td>Client library version is out of sync with DSE.</td>
</tr>
<tr>
<td>RTLAPI_BAD_GROUP</td>
<td>group_name is null or longer than 32 characters.</td>
</tr>
<tr>
<td>RTLAPI_BAD_STORESYSNAME</td>
<td>storagesys_name is longer than 256 characters.</td>
</tr>
<tr>
<td>RTLAPI_BADSTORENAME</td>
<td>store_name is longer than 256 characters.</td>
</tr>
<tr>
<td>RTLAPI_MALLOC_FAILED</td>
<td>Either DSE or the client API could not allocate space.</td>
</tr>
<tr>
<td>RTLAPI_IN_SCAN</td>
<td>rtl_attach was called at the incorrect time.</td>
</tr>
<tr>
<td>RTLAPI_BAD_SVCNAME</td>
<td>service_name is null and RTL_SERVICENAME environment variable not found.</td>
</tr>
<tr>
<td>RTLAPI_STORESYS_LOAD_ERR</td>
<td>Could not load shared library implementing the storage system.</td>
</tr>
<tr>
<td>RTLAPI_STORESYS_OPEN_ERR</td>
<td>Could not open the storage system.</td>
</tr>
<tr>
<td>RTLAPI_STORE_OPEN_ERR</td>
<td>Could not open the store.</td>
</tr>
<tr>
<td>RTLAPI_ATTACH_REFUSED</td>
<td>Attach not allowed to:</td>
</tr>
<tr>
<td></td>
<td>• A standby DSE instance.</td>
</tr>
<tr>
<td></td>
<td>• A store using a simple schema.</td>
</tr>
<tr>
<td>RTLAPI_SOCK_OPEN_ERR</td>
<td>Could not open socket to DSE.</td>
</tr>
<tr>
<td>RTLAPI_SOCK_CONNECT_ERR</td>
<td>Could not connect on socket to DSE.</td>
</tr>
<tr>
<td>RTLAPI_SOCK_OPTION_ERR</td>
<td>Could not set socket options.</td>
</tr>
</tbody>
</table>

Related concepts

"C client API" on page 285

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_begin_scan()**

To retrieve a range of data from DSE, you must first start a scan. The scan descriptor determines what data will be returned. The **rtl_begin_scan()** function allows you to select all or a subset of the symbols in the store and a time range.

### Syntax

```c
rtlscan *rtl_begin_scan(
    rtl_desc *rdesc,
    char *name,
    rtl_datetime *start,
    rtl_datetime *end,
    int flags,
    int *err)
```

### Arguments

- **rdesc** Points to the `rtl_desc` structure that is allocated by the `rtl_attach` or `rtl_reattach` function.
- **name** Identifies the symbol or symbols for which you want to retrieve data.

  *name* can have one of three different formats. If **name** is null, the function scans all symbols for the attached store. If **name** is a single symbol name, the function scans only that symbol. If **name** is a set of symbols, the function scans only the symbols in the set. The format of the set string is a list of names that are separated by commas and enclosed in single quotation marks.

  **Example:** ‘AAA’, ‘BBB’, ‘CCC’ scans the symbols AAA, BBB, and CCC. You must enclose each symbol name in single quotation marks.

- **start** Denotes where the scan starts.

  The scans start where the data has a timestamp greater than or equal to the value of **start**. If **start** is null, the scan begins with the earliest data.

- **end** Denotes where the scan ends.

  The scan ends where the data has a timestamp less than or equal to the value of **end**. If **end** is null, the scan ends with the most recent data.

- **flags** Denotes any special instructions for the scan.

  **flags** is a bit map with the following bits:

  - **RTLSCAN_SKIP_START**
    
    Look for data with timestamps greater than **start** instead of greater than or equal to **start**.

  - **RTLSCAN_SKIP_END**
    
    Look for data with timestamps less than **end** instead of less than or equal to **end**.

  - **RTLSCAN_INMEM_ONLY**
    
    Look for data in memory only.

  - **RTLSCAN_STORE_ONLY**
    
    Look for data in the store only.
**RTLSCAN_SET**

Look for data for multiple symbols; *name* is an SQL set expression.

**Restriction:** RTLSCAN_INMEM_ONLY and RTLSCAN_STORE_ONLY are mutually exclusive; you can specify at most one of these options.

**err**

Return code status.

If you set err as null, the function returns -1.

**Return values**

If rtl_begin_scan() is successful, it sets the err argument to RTLAPI_SUCCESS.

If rtl_begin_scan() fails, it sets the err argument to one of the following return codes:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_END_B4_START</td>
<td>End date is before the start date.</td>
</tr>
<tr>
<td>RTLAPI_BAD_SCANFLAGS</td>
<td><em>flags</em> has an undefined bit set or a bad combination of bits.</td>
</tr>
<tr>
<td>RTLAPI_BAD_DESC</td>
<td><em>rdesc</em> is null or points to an invalid rtldesc structure.</td>
</tr>
<tr>
<td>RTLAPI_IN_SCAN</td>
<td>Already in a scan.</td>
</tr>
<tr>
<td>RTLAPI_MALLOC_FAILED</td>
<td>Either DSE or the client API could not allocate space.</td>
</tr>
<tr>
<td>RTLAPI_BAD_REGEXP</td>
<td>Regular expression is bad.</td>
</tr>
<tr>
<td>RTLAPI_BAD_LIKE_EXPR</td>
<td>Like expression is bad.</td>
</tr>
<tr>
<td>RTLAPI_BAD_SET_EXPR</td>
<td>Set expression is bad.</td>
</tr>
<tr>
<td>RTLAPI_STORE_OFFLINE</td>
<td>Store is offline.</td>
</tr>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Connection to DSE failed or dropped.</td>
</tr>
<tr>
<td>RTLAPI_STORE_ERROR</td>
<td>A storage system error occurred. Call the rtl_end_scan function to terminate the query, then call the rtl_begin_scan function to restart the query.</td>
</tr>
<tr>
<td>RTLAPI_NOT_CONNECTED</td>
<td>Not connected to a store.</td>
</tr>
<tr>
<td>RTLAPI_BAD_NAME</td>
<td><em>name</em> is null or an empty string.</td>
</tr>
</tbody>
</table>

**Related concepts**

[C client API](#) on page 285

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_client_log()**

This function provides a way for messages to be generated from a file of numbered error messages.

The messages can contain parameter placeholders in the standard printf format notation, which will be populated based on the variable argument list passed to the function.

**Syntax**

```
char * rtl_client_log(
    char **msg_desc,
    int p_errno, ...
)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **msg_desc**
  The message descriptor that is returned by the rtl_client_log_init() function.

- **p_errno**
  The error number from the message file that should be printed.

**Return values**

The message represented by the error number with all placeholder variables completed based on the variable argument list.

**Note:** The memory buffer for the message string is allocated by this function and should be freed by the caller after use.
**rtl_client_log_done()**

This function shuts down the DSE client logging functionality and frees all resources allocated by the rtl_client_log_init().

After calling the rtl_client_log_done() function, the rtl_client_log_init() function must be called again before calls can be made to the rtl_client_log() function.

**Syntax**

```c
void rtl_client_log_done(
    char **msg_desc);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*msg_desc*

The error message descriptor returned by the rtl_client_log_init() function.

**Return values**

The rtl_client_log_done() function has no return values.
rtl_client_log_init()

This function initializes the DSE client logging functionality.

No calls should be made to rtl_client_log without first calling the rtl_client_log_init() function. The rtl_client_log_done() function should be called to shutdown the DSE client logging functionality and free all resources allocated by this function.

Syntax

```
char **rtl_client_log_init(
    char *errfile);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

errfile  The path of the file of error messages to be used for client logging.

Return values

If the rtl_client_log_init() function is successful, it returns an error message descriptor. This descriptor is passed to all subsequent calls to the rtl_client_log() function and the rtl_client_log_done() function.

If the rtl_client_log_init() function fails, it returns NULL.
**rtl_detach()**

This function disconnects a C client application from the data store and from DSE. This function detaches the connection that was established by the rtl_attach() function.

**Syntax**

```c
void rtl_detach(
    rtldesc *rdesc)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **rdesc** A valid rtldesc that is returned by rtl_attach or rtl_reattach.

**Return values**

This function has no return values.

**Related concepts**

[C client API” on page 285](#)

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
rtl_elem()

This function retrieves a tick record for a point in time from the data store.

Syntax

```c
int rtl_elem(
    rtldesc *rdesc,
    char *name,
    rtl_datetime *stamp,
    int cmp,
    void **ret_tick,
    char **nulls)
```

Arguments

- **rdesc** A valid `rtldesc` that is returned by `rtl_attach` or `rtl_reattach`.
- **name** Name of the symbol. `name` can only identify a single symbol.
- **stamp** Timestamp to use to locate the data.
- **cmp** Determines how the `stamp` argument is treated.
  - RTL_LT: Looks for the first data with a timestamp of less than the value of `stamp`.
  - RTL_LE: Looks for the first data with a timestamp of less than or equal to the value of `stamp`.
  - RTL_EQ: Looks for the first data with a timestamp equal to the value of `stamp`.
  - RTL_GE: Looks for the first data with a timestamp of greater than or equal to the value of `stamp`.
  - RTL_GT: Looks for the first data with a timestamp or greater than the value of `stamp`.
- **ret_tick** Returns a pointer to the new record.
  - If there is an error, the return value is undefined.
- **nulls** Returns a pointer to an array of null value indicators.
  - There is one entry in the array for each field in `ret_tick`. If an entry is set to zero, the corresponding field is not null; otherwise the field is null. If there is an error, the return value is undefined. The caller must not free this array.
Return values

If rtl_elem() is successful, it returns RTLAPI_SUCCESS.

If rtl_elem() fails, it returns one of the following return codes:

Table 42. Return Codes

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_NO_DATA</td>
<td>No data qualified with the given arguments.</td>
</tr>
<tr>
<td>RTLAPI_BAD_DESC</td>
<td>rtldesc is null or points to an invalid rtldesc structure.</td>
</tr>
<tr>
<td>RTLAPI_BAD_RET_TICK</td>
<td>ret_tick is null.</td>
</tr>
<tr>
<td>RTLAPI_BAD_NULLS</td>
<td>nulls is null.</td>
</tr>
<tr>
<td>RTLAPI_IN_SCAN</td>
<td>Already in a scan.</td>
</tr>
<tr>
<td>RTLAPI_MALLOC_FAILED</td>
<td>Either DSE or the client API could not allocate space.</td>
</tr>
<tr>
<td>RTLAPI_NOT_CONNECTED</td>
<td>DSE is not connected to a store.</td>
</tr>
<tr>
<td>RTLAPI_BAD_STAMP</td>
<td>stamp is null.</td>
</tr>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Communication error with DSE.</td>
</tr>
<tr>
<td>RTLAPI_STORE_ERROR</td>
<td>A storage system error occurred. Retry the operation later.</td>
</tr>
<tr>
<td>RTLAPI_BAD_NAME</td>
<td>name is null.</td>
</tr>
<tr>
<td>RTLAPI_BAD_CMP</td>
<td>cmp is not a valid comparison type.</td>
</tr>
</tbody>
</table>

Related concepts

"C client API" on page 285
DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
rtl_end_scan()

This function ends a scan for tick data that was started with the rtl_begin_scan() function.

Syntax

```c
int rtl_end_scan(
    rtlscan *rscan)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

rscan  A valid rscan pointer returned by rtl_begin_scan that rtl_end_scan uses to free all associated structures. On return, it is not possible to use the scan pointer to retrieve any more data.

Return values

If rtl_end_scan() is successful, it returns RTLAPI_SUCCESS.

If rtl_end_scan() fails, it returns one of the following return codes:

Table 43. Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Communication error with DSE.</td>
</tr>
<tr>
<td>RTLAPI_NOT_IN_SCAN</td>
<td>Not currently in a scan.</td>
</tr>
<tr>
<td>RTLAPI_CORRUPT_DATA</td>
<td>Data buffer received from DSE was corrupted.</td>
</tr>
<tr>
<td>RTLAPI_BAD_DESC</td>
<td>rscan is null or points to an invalid rtlscan structure.</td>
</tr>
</tbody>
</table>

Related concepts

"C client API" on page 285

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_feed_get_gapfill()**

This function returns the gapfill for the feed.

The gapfill is the value of the FEEDXX_GAPFILL parameter in the configuration file, where XX is the feed ID that is passed to the function.

**Syntax**

```c
int rtl_feed_get_gapfill(
    int p_feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- `p_feed_id`
  - The ID number of the feed

**Return values**

If rtl_feed_get_gapfill is successful, it returns:

- 0  If the feed is not a gapfill feed.
- 1  If the feed is a gapfill feed.

If rtl_feed_get_gapfill() fails, it returns RTL_FAILURE.

**Related concepts**

"Feed handler API" on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_feed_get_handler_id()**

This function returns the handler ID for the feed.

The handler ID corresponds to the FEEDXX_HANDLER_NAME parameter in the configuration file, where XX is the feed ID that is passed to the function.

**Syntax**

```c
int rtl_feed_get_handler_id(
    int p_feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **p_feed_id**
  The ID number of the feed

**Return values**

If `rtl_feed_get_handler_id()` is successful, it returns the handler ID.

If `rtl_feed_get_handler_id()` fails, it returns RTL_FAILURE.

**Related concepts**

"Feed handler API" on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_feed_get_handler_name()**

This function returns the handler name for the feed.

The handler name is the value of the FEEDXX_HANDLER_NAME parameter in the configuration file, where XX is the feed ID that is passed to the function.

**Syntax**

```c
char * rtl_feed_get_handler_name(
    int p_feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`p_feed_id`

The ID number of the feed

**Return values**

If `rtl_feed_get_handler_name()` is successful, it returns the handler name.

If `rtl_feed_get_handler_name()` fails, it returns a zero-length string.

If the feed specified by the feed ID is not configured, `rtl_feed_get_handler_name()` returns a zero-length string. You can use the `rtl_feed_is_configured()` function to determine if a feed is configured for a particular feed ID.

**Related concepts**

"Feed handler API" on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.
rtl_feed_get_info()

This function returns the feed information for the feed.

The feed information is the value of the FEEDXX_INFO parameter in the configuration file, where XX is the feed ID that is passed to the function.

Syntax

```c
char * rtl_feed_get_info(
    int p_feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

p_feed_id

The ID number of the feed

Return values

If rtl_feed_get_info() is successful, it returns a pointer to the feed information string.

If rtl_feed_get_info() fails, it returns a zero-length string.

If the feed specified by the feed ID is not configured, this function returns a zero-length string. You can use the rtl_feed_is_configured() function to determine if a feed for a particular feed ID is configured.

Note: The FEEDXX_INFO config parameter is optional. If it is not provided, a zero-length string is returned.

Related concepts

"Feed handler API" on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_feed_get_source()**

This function returns the current source for the feed.

The initial source is specified by the FEEDXX_SOURCE parameter in the configuration file; the source changes whenever a source switch occurs.

**Syntax**

```c
char * rtl_feed_get_source(
    int p_feed_id);
```

**Arguments**

*p_feed_id*  
The ID number of the feed.

**Return values**

If rtl_feed_get_source() is successful, it returns the current source.

If rtl_feed_get_source() fails, it returns a zero-length string.

If the feed specified by the feed ID is not configured, this function returns a zero-length string. You can use the rtl_feed_is_configured() function to determine if a feed for a particular feed ID is configured.

**Related concepts**

[DSE provides API functions for feed handlers to retrieve information about the feed.](#)
rtl_feed_get_store_id()

This function returns the store ID for the feed.

The store ID corresponds to the FEEDXX_STORE_NAME parameter in the configuration file, where XX is the feed ID that is passed to the function.

Syntax

```c
int rtl_feed_get_store_id(
    int p_feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

**p_feed_id**

The ID number of the feed.

Return values

If rtl_feed_get_store_id() is successful, it returns the store ID.

If rtl_feed_get_store_id() fails, it returns RTL_FAILURE.

If the feed specified by the feed ID is not configured, this function returns RTL_FAILURE. You can use the rtl_feed_is_configured() function to determine if a feed for a particular feed ID is configured.

**Note:** The FEEDXX_STORE_NAME configuration parameter is optional. If it is not provided, a value of -1 (RTL_FAILURE) is returned.

Related concepts

"Feed handler API" on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_feed_get_store_name()**

This function returns the store name for the feed.

The store name is the value of the FEEDXX_STORE_NAME parameter in the configuration file, where XX is the feed ID that is passed to the function.

**Syntax**

```c
char * rtl_feed_get_store_name(
    int p_feed_id);
```

**Valid in client application?** | **Valid in storage system?** | **Valid in feed handler?** | **Valid in HA library?** | **Valid in transport?**
--- | --- | --- | --- | ---
No | No | Yes | No | No

**Arguments**

**p_feed_id**

The ID number of the feed.

**Return values**

If `rtl_feed_get_store_name()` is successful, it returns the store name.

If `rtl_feed_get_store_name()` fails, it returns a zero-length string.

If the feed specified by the feed ID is not configured, this function returns a zero-length string. You can use the `rtl_feed_is_configured()` function to determine if a feed for a particular feed ID is configured.

**Note:** The FEEDXX_STORE_NAME config parameter is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

“Feed handler API” on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_feed_get_target()**

This function returns the current target for the feed.

The initial target is specified by the FEEDXX_TARGET parameter in the configuration file; the target changes whenever a target switch occurs.

**Syntax**

```c
char * rtl_feed_get_target(
    int p_feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

**p_feed_id**

The ID number of the feed.

**Return values**

If `rtl_feed_get_target()` is successful, it returns the current target.

If `rtl_feed_get_target()` fails, it returns a zero-length string.

If the feed specified by the feed ID is not configured, this function returns a zero-length string. You can use the `rtl_feed_is_configured()` function to determine if a feed for a particular feed ID is configured.

**Note:** The FEEDXX_TARGET configuration parameter is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

- "Feed handler API" on page 287
- DB2 Data Stream Engine User’s Guide and Reference
**rtl_feed_get_transport_id()**

The rtl_feed_get_transport_id() function returns a transport ID for the corresponding feed ID.

**Syntax**

```c
int rtl_feed_get_transport_id(int feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

**feed_id (in)**

The ID of the feed.

**Return values**

If rtl_feed_get_transport_id() is successful, it returns the transport ID.

If rtl_feed_get_transport_id() fails, it returns RTL_FAILURE.

If the transport specified by `feed_id` is not configured, rtl_feed_get_transport_id() returns RTL_FAILURE. You can use the rtl_transport_is_configured() function to determine if a transport for a particular transport ID is configured.

**Related concepts**

"Feed handler API" on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.
rtl_feed_get_user_data()
This function returns the feed user data for the feed. The feed user data is the memory buffer allocated by the feed handler and is returned to DSE from the rtl_feed_init() call in the feed handler implementation. The contents and use of that area are determined by the feed handler implementation.

Syntax
void * rtl_feed_get_user_data(
    int p_feed_id);

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments
p_feed_id
The ID number of the feed.

Return values
If rtl_feed_get_user_data() is successful, it returns a pointer to the feed user data area.

If rtl_feed_get_user_data() fails or no user data has been allocated, it returns null.

Related concepts
"Feed handler API" on page 287
DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_feed_is_configured()**

This function enables you to determine if a particular feed is defined in the configuration file. You can use this function to check the validity of a feed ID before calling other feed accessor functions, such as the rtl_feed_get_info() function.

**Syntax**

```c
int rtl_feed_is_configured(
    int p_feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **p_feed_id**
  The ID number of the feed.

**Return values**

The rtl_feed_is_configured() function returns 1 if the feed for the specified ID is configured. Otherwise, it returns 0.

**Related concepts**

"Feed handler API" on page 287
DSE provides API functions for feed handlers to retrieve information about the feed.
rtl_feed_log()

This function provides a means for messages to be displayed in the DSE message log. The message file that a feed uses is defined by the value of the HANDLERXX_MSG_PATH configuration parameter for the feed’s corresponding HANDLER entry.

Syntax

```c
void rtl_feed_log(
    int feedno,
    int p_errno,
    ...);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

feedno

The ID number of the feed.

p_errno

The error number from the message file that should be printed.

... A printf-style variable-length list of arguments corresponding to parameters in the log message.

Return values

The rtl_feed_log() function has no return values.
**rtl_feed_source_get_message()**

This function reads a message from the feed data source.

This function should only be called by the feed thread that corresponds to the feed ID that is passed to the function.

**Syntax**

```c
int rtl_feed_source_get_message(
    int feed_id,
    unsigned char **msg_buf,
    int *msg_len);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*feed_id (in)*

The ID of the feed.

*msg_buf (in/out)*

A return pointer to a buffer containing a full message from the data source. Start and end delimiters, if present in the incoming data stream, are not included in this buffer. The transport is responsible for allocating and freeing this buffer. The caller should copy any data that should be kept.

*msg_len (out)*

The length, in bytes, of the message that is returned in the *msg_buf* argument.

**Return values**

If this function is successful, it returns RTL_SUCCESS.

If this function fails, it returns RTL_FAILURE.
**rtl_feed_source_put_message()**

This function writes a message to the feed data source.

This function should be called only by the feed thread that corresponds to the feed ID that is passed to the function.

**Syntax**

```c
int rtl_feed_source_put_message(
    int feed_id,
    unsigned char *msg_buf,
    int msg_len);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **feed_id (in)**
  
The ID of the feed.

- **msg_buf (in)**
  
The message buffer to be written to the feed data source. If length information or delimiters are specified in the source_msgfmt argument to `rtl_transport_source_open()`, the data that is written to the source is preceded and followed by the appropriate bytes.

- **msg_len (in)**
  
The length of the message data in the message buffer.

**Return values**

If this function is successful, it returns RTL_SUCCESS.

If this function fails, it returns RTL_FAILURE.
**rtl_feed_state_change_register()**

This function registers state change function for a feed.

DSE calls the given state change function for each state change for the feed. This function should be called only by the feed thread that corresponds to the feed ID that is passed to the function.

**Syntax**

```c
int rtl_feed_state_change_register(
    int feed_id,
    void (*state_change_func)(int, void*, rtl_feed_state_t));
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

**feed_id (in)**

The ID of the feed.

**state_change_func (in)**

The feed handler’s state change function to be registered with DSE. The arguments for this state change function are:

- The feed ID
- The feed user data that is allocated and initialized by rtl_feed_init
- The feed state

The feed state can be one of the following:

**RTL_FEED_STATE_SHUTDOWN**

The feed was shut down.

**Note:** The feed user data can be null.

**RTL_FEED_STATE_INITED**

The feed was initialized.

**RTL_FEED_STATE_IDLE**

The feed is waiting for a new data source.

**RTL_FEED_STATE_PAUSED**

The feed was paused using the rtm mode -fp command.

**RTL_FEED_STATE_RUNNING**

The feed was started after being initialized, resuming from being paused or being given a new data source.

**RTL_FEED_STATE_BLOCKED**

The feed was blocked from writing to shared memory and is waiting for free space in the tick pool.

**RTL_FEED_STATE_SOURCE_OPEN**

The feed opened the feed data source.

**RTL_FEED_STATE_SOURCE_CLOSE**

The feed closed the feed data source.

**RTL_FEED_STATE_TARGET_OPEN**

The feed opened the feed data target.
RTL_FEED_STATE_TARGET_CLOSE
The feed closed the feed data target.

Return values
If this function is successful, it returns RTL_SUCCESS.

If this function fails, it returns RTL_FAILURE.
**rtl_get_feed_for_symbol()**

This function gets the feed ID that is responsible for populating a given symbol in a given store.

**Syntax**

```c
int rtl_get_feed_for_symbol(
    const char *p_symbol,
    int p_store_id,
    int *p_feed_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **p_symbol**
  - The symbol name (input parameter).

- **p_store_id**
  - The store ID for the symbol (input parameter).

- **p_feed_id**
  - The feed ID that is responsible for populating the symbol in the store (output parameter).

**Return values**

If the rtl_get_feed_for_symbol() function is successful, it returns RTL_SUCCESS.

If the rtl_get_feed_for_symbol() function fails, it returns RTL_FAILURE.

If the store ID is invalid, the function returns RTL_INVALID_STORE.

If the symbol cannot be found, the function returns RTL_SYMBOL_NOT_FOUND.
**rtl_get_metadata()**

This function reads metadata from shared memory for a specified symbol. This function returns the entire metadata structure for the specified symbol.

**Syntax**

```c
int rtl_get_metadata(
    rtldesc *rdesc,
    char *name,
    void **ret_meta,
    char **nulls);
```

**Arguments**

- **rdesc**  
  A valid `rtldesc` returned by `rtl_attach` or `rtl_reattach`.
- **name**  
  Name of the symbol. `name` can identify only a single symbol.
- **ret_meta**  
  Returns a pointer to the metadata that is returned.  
  If there is an error, `ret_meta` is not set.
- **nulls**  
  A return pointer to an array of null indicators.  
  There is one entry in the array for each field in `ret_meta`. If an entry is set to zero, the corresponding field is not null; otherwise the field is null. If there is an error, the return value is undefined. The caller must not free this array.

**Return values**

If `rtl_get_metadata()` is successful, it returns `RTLAPI_SUCCESS`.

If `rtl_get_metadata()` fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_BAD_DESC</td>
<td><code>rdesc</code> is null or points to an invalid <code>rtldesc</code> structure.</td>
</tr>
<tr>
<td>RTLAPI_NO_SUCH_NAME</td>
<td><code>name</code> not found in the store.</td>
</tr>
<tr>
<td>RTLAPI_MALLOC_FAILED</td>
<td>Either DSE or the client API could not allocate space.</td>
</tr>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Communication error with DSE.</td>
</tr>
<tr>
<td>RTLAPI_BAD_RETMETA</td>
<td><code>ret_meta</code> is null.</td>
</tr>
<tr>
<td>RTLAPI_BAD_NULLS</td>
<td><code>nulls</code> is null.</td>
</tr>
<tr>
<td>RTLAPI_BAD_NAME</td>
<td><code>name</code> is null.</td>
</tr>
<tr>
<td>RTLAPI_NOT_CONNECTED</td>
<td>Not connected to a store.</td>
</tr>
<tr>
<td>RTLAPI_NULLIND_ERR</td>
<td>Could not obtain null indicators.</td>
</tr>
<tr>
<td>RTLAPI_IN_SCAN</td>
<td>Already in a scan.</td>
</tr>
</tbody>
</table>

**Related concepts**
DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_get_meta_info()**

This function gets information about the metadata for a record type.

The metadata is defined by the RECORDXX_META_SCHEMA parameter in the configuration file, where XX is the record ID that is passed to the function.

**Syntax**

```c
int rtl_get_meta_info(
    int rectype,
    int *size,
    int *fieldcnt,
    field_struct_t **fields,
    meta_field_struct_t **metafields);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

rectype

The numeric ID of the record (input parameter).

size

The size of the metadata record in bytes (output parameter).

fieldcnt

The number of fields in the metadata record (output parameter).

fields

An array of field information for each field in the metadata record (output parameter).

metafields

An array of metadata-specific field information for each field in the metadata record (output parameter).

**Return values**

If the rtl_get_meta_info() function is successful, it returns RTL_SUCCESS.

If the rtl_get_meta_info() function fails, it returns RTL_FAILURE.

**Related concepts**

- [“Record type API” on page 289](#)
  - DSE provides API functions for retrieving information about record types and setting default values for record types.
- [Chapter 10, “Supported data types,” on page 83](#)
  - DSE provides or interacts with different data type systems that developers can use when working with different kinds of data streams and storage systems.
**rtl_get_meta_rec_info()**

This function retrieves information about the metadata structure that is associated with the *rtldesc* that you specify.

**Syntax**

```c
int rtl_get_meta_rec_info(
    rtldesc *rdesc,
    int *size,
    int *fieldcnt,
    field_struct_t **fields,
    meta_field_struct_t **metafields)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **rdesc** A valid *rtldesc* that is returned by rtl_attach or rtl_reattach.
- **size** Size in bytes of the record.
- **fieldcnt** Number of fields in the record.
- **fields** An array that holds information about each field in the record. See the rtl_rectype.h file for the content of this structure.
- **metafields** An array with additional metadata-specific information about each field in the record.

If the value of *size, fieldcnt, fields, or metafields* is a null pointer, no corresponding information is returned to the caller.

**Return values**

If rtl_get_meta_rec_info() is successful, it returns RTLAPI_SUCCESS.

If rtl_get_meta_rec_info() fails because *rdesc* is null or points to an invalid rtldesc structure, this function returns RTLAPI_BAD_DESC.

**Related concepts**

[C client API](#) on page 285

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_get_next()**

This function retrieves the next tick record for a scan query.

You must call the rtl_begin_scan function before calling this function.

**Syntax**

```c
int rtl_get_next(
    rtlscan *rscan,
    void **ret_tick,
    char **nulls,
    char **name)
```

**Arguments**

- **rscan**  A valid rtlscan pointer returned by the rtl_begin_scan function.
- **ret_tick**  Returns a pointer to the new record. If there is an error, the return value is undefined.
- **nulls**  Returns a pointer to an array of null value indicators. There is one entry in the array for each field in ret_tick. If an entry is set to zero, the corresponding field is not null; otherwise the field is null. If there is an error, the return value is undefined. The caller must not free this array.
- **name**  Returns a pointer to the name of the symbol. If an error occurs, the returned symbol name is undefined. You can call rtl_get_next with name set to null, but the function will not return a value.

**Return values**

If rtl_get_next() is successful, it returns RTLAPI_SUCCESS.

If rtl_get_next() fails, it returns one of the following return codes:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_NO_MORE_RESULTS</td>
<td>No more data to return.</td>
</tr>
<tr>
<td>RTLAPI_BAD_DESC</td>
<td>rscan is null or points to an invalid rtlscan structure.</td>
</tr>
<tr>
<td>RTLAPI_BAD_RETTICK</td>
<td>ret_tick is null.</td>
</tr>
<tr>
<td>RTLAPI_BAD_NULLS</td>
<td>nulls is null.</td>
</tr>
<tr>
<td>RTLAPI_MALLOC_FAILED</td>
<td>Either DSE or the client API could not allocate space.</td>
</tr>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Connection to DSE failed or dropped.</td>
</tr>
<tr>
<td>RTLAPI_STORE_ERROR</td>
<td>A storage system error occurred. Call the rtl_end_scan function to terminate the query, then call the rtl_begin_scan function to restart the query.</td>
</tr>
<tr>
<td>RTLAPI_IN_SCAN</td>
<td>Already in a scan.</td>
</tr>
</tbody>
</table>
Table 45. Return Codes  (continued)

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_CORRUPT_DATA</td>
<td>Data buffer received from DSE was corrupted.</td>
</tr>
</tbody>
</table>

Related concepts

C client API on page 285
DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_get_rectype_id_from_name()**

This function returns the record type ID for the record.

The record ID is the value of XX in the RECORDXX _NAME parameter in the configuration file.

**Syntax**

```c
int rtl_get_rectype_id_from_name(
    char *name);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **name**  The record name.

**Return values**

If the rtl_get_rectype_id_from_name() function is successful, it returns the record ID.

If the rtl_get_rectype_id_from_name() function fails or cannot find a record with the given name, it returns RTL_FAILURE.

**Related concepts**

- "Record type API" on page 289
  
  DSE provides API functions for retrieving information about record types and setting default values for record types.
**rtl_get_rtltime_date_fields()**

This function gets the date components of an rtl_datetime type.

**Syntax**

```c
void rtl_get_rtltime_date_fields(
    rtl_datetime *rtl_time,
    int *year,
    int *month,
    int *day);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_time**
  The timestamp as an rtl_datetime type (input parameter).
- **year**
  The year component of the rtl_datetime timestamp (output parameter).
- **month**
  The month component of the rtl_datetime timestamp (output parameter).
- **day**
  The day component of the rtl_datetime timestamp (output parameter).

**Return values**

The rtl_get_rtltime_date_fields() function has no return values.

**Related concepts**

"Time API" on page 291

DSE provides API functions for managing timestamps.
**rtl_get_rtltime_fields()**

This function gets the date and time components of an rtl_datetime type.

**Syntax**

```c
void rtl_get_rtltime_fields(
    rtl_datetime *rtl_time,
    int *year,
    int *month,
    int *day,
    int *hour,
    int *minute,
    int *second,
    int *microsecond);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_time**
  The timestamp as an rtl_datetime type (input parameter).
- **year**
  The year component of the rtl_datetime timestamp (output parameter).
- **month**
  The month component of the rtl_datetime timestamp (output parameter).
- **day**
  The day component of the rtl_datetime timestamp (output parameter).
- **hour**
  The hour component of the rtl_datetime timestamp (output parameter).
- **minute**
  The minute component of the rtl_datetime timestamp (output parameter).
- **second**
  The second component of the rtl_datetime timestamp (output parameter).
- **microsecond**
  The microsecond component of the rtl_datetime timestamp (output parameter).

**Return values**

The rtl_get_rtltime_fields() function has no return values.

**Related concepts**

["Time API" on page 291]

DSE provides API functions for managing timestamps.
rtl_get_rtltime_time_fields()

This function gets the time components of an rtl_datetime type.

**Syntax**

```c
void rtl_get_rtltime_time_fields(
    rtl_datetime *rtl_time,
    int    *hour,
    int    *minute,
    int    *second);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_time**
  The timestamp as an rtl_datetime type (input parameter).

- **hour**
  The hour component of the rtl_datetime timestamp (output parameter).

- **minute**
  The minute component of the rtl_datetime timestamp (output parameter).

- **second**
  The second component of the rtl_datetime timestamp (output parameter).

**Return values**

The rtl_get_rtltime_time_fields() function has no return values.

**Related concepts**

- "Time API" on page 291
- DSE provides API functions for managing timestamps.
**rtl_get_tick_info()**

This function gets information about the tick for a record type.

The tick is defined by the RECORDXX_SCHEMA parameter in the configuration file, where XX is the record ID that is passed to the function.

**Syntax**

```c
int rtl_get_tick_info(
    int rectype,
    int *size,
    int *fieldcnt,
    field_struct_t **fields,
    tick_field_struct_t **tickfields);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **rectype**
  - The numeric ID of the record (input parameter).
- **size**
  - The size of the tick record in bytes (output parameter).
- **fieldcnt**
  - The number of fields in the tick record (output parameter).
- **fields**
  - An array of field information for each field in the tick record (output parameter).
- **tickfields**
  - An array of tick-specific field information for each field in the tick record (output parameter).

**Return values**

If the rtl_get_tick_info() function is successful, it returns RTL_SUCCESS.

If the rtl_get_tick_info() function fails, it returns RTL_FAILURE.

**Related concepts**

- "Record type API” on page 289
  DSE provides API functions for retrieving information about record types and setting default values for record types.
- Chapter 10, “Supported data types,” on page 83
  DSE provides or interacts with different data type systems that developers can use when working with different kinds of data streams and storage systems.
rtl_get_tick_rec_info()

This function retrieves information about the tick data structure that is associated with the rtldesc that you specify.

Syntax

```c
int rtl_get_tick_rec_info(
    rtldesc *rdesc,
    int *size,
    int *fieldcnt,
    field_struct_t **fields,
    tick_field_struct_t **tickfields)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

rdesc  A valid rtldesc that is returned by rtl_attach or rtl_reattach.

size  Size in bytes of the record.

fieldcnt  Number of fields in the record.

fields  An array with information about each field in the record. See the rtl_rectype.h file for the content of this structure.

tickfields  An array with additional tick-specific information about each field in the record.

If the value of size, fieldcnt, fields, or tickfields is a null pointer, no corresponding information is returned to the caller.

Return values

If rtl_get_tick_rec_info() is successful, it returns RTLAPI_SUCCESS.

If rtl_get_tick_rec_info() fails because rdesc is null or points to an invalid rtldesc structure, this function returns RTLAPI_BAD_DESC.

Related concepts

[C client API](#) on page 285

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
rtl_get_udp_value()

This function returns the user-defined parameter (UDP) for the given UDP ID.

The UDP ID is the XX value of the UDPXX parameter in the configuration file.

UDP parameters can be string or integer values. User-defined parameters enclosed by quotation marks are assumed to be strings; parameters not enclosed by quotation marks are assumed to be integers.

**Syntax**

```c
void *rtl_get_udp_value(
    int p_offset);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

p_offset

The numeric ID of the User-defined parameter.

**Return values**

This function returns either an int or a string value, depending on what type was set in the configuration file.
**rtl_gmtime_to_rtltime()**

This function generates an rtl_datetime type timestamp for the current GMT time.

The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
void rtl_gmtime_to_rtltime(
    rtl_datetime *rtl_time);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_time**
  
  The current GMT time as an rtl_datetime type (output parameter).

**Return values**

The rtl_gmtime_to_rtltime() function has no return values.

**Related concepts**

- "Time API" on page 291

  DSE provides API functions for managing timestamps.
rtl_ha_get_peer_id_from_name()

This function returns a numeric ID value for the specified peer DSE instance.

The ID is the value of XX in the HAXX_PEER_NAME parameter in the configuration file.

**Syntax**

```c
int rtl_ha_get_peer_id_from_name(char *peer_name);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Parameters**

**peer_name (in)**

The name of the peer DSE instance.

**Return values**

If none of the HAXX_PEER_NAME entries has the value provided in the peer_name argument, this function returns -1.

**Related concepts**

“High availability information accessor API” on page 288

DSE provides API functions for the high availability communication library to use. The purpose of these API functions is to retrieve information about the high availability configuration.
rtl_ha_get_peer_info_from_id()

This function returns the information about the peer instance that corresponds to the specified peer ID.

The returned information is the value of the HAXX_PEER_INFO parameter in the configuration file.

**Syntax**

```
char *rtl_ha_get_peer_info_from_id(int peer_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Parameters**

**peer_id (in)**

The numeric peer ID of the peer DSE instance that the peer information is requested for.

**Return values**

If the specified peer_id is invalid, this function returns null.

**Related concepts**

“High availability information accessor API” on page 288

DSE provides API functions for the high availability communication library to use. The purpose of these API functions is to retrieve information about the high availability configuration.
rtl_ha_get_peer_name_from_id()

This function returns the name of the peer instance that corresponds to the specified peer ID.

The name of the peer instance is the value of the HAXX_PEER_NAME parameter in the configuration file.

Syntax

cchar *rtl_ha_get_peer_name_from_id(int peer_id);

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Parameters

peer_id (in)

The numeric peer ID of the peer DSE instance that the peer name is requested for.

Return values

If the specified peer_id is invalid, this function returns null.

Related concepts

“High availability information accessor API” on page 288

DSE provides API functions for the high availability communication library to use. The purpose of these API functions is to retrieve information about the high availability configuration.
`rtl_ha_log()`

The `rtl_ha_log` function enables you to display messages in the DSE message log. The message file that a high availability library uses is defined by the HA_MSG_PATH parameter in the configuration file.

**Syntax**

```c
void rtl_ha_log(
    int p_errno,
    ...);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **p_errno**
  - The error number from the message file that should be printed.

- **...**
  - A printf-style variable-length list of arguments corresponding to parameters in the log message.

**Return values**

The `rtl_ha_log()` function has no return values.

**Related concepts**

"High availability information accessor API" on page 288

DSE provides API functions for the high availability communication library to use. The purpose of these API functions is to retrieve information about the high availability configuration.
**rtl_handler_get_id_from_name()**

This function returns the handler ID for the feed handler.

The handler ID is the value of XX in the HANDLERXX_NAME parameter in the configuration file.

**Syntax**

```c
int rtl_handler_get_id_from_name(
    char *p_name);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`p_name`

The feed handler name

**Return values**

If `rtl_handler_get_id_from_name()` is successful, it returns the handler ID.

If `rtl_handler_get_id_from_name()` fails or cannot find a handler with the given name, it returns RTL_FAILURE.

**Related concepts**

*Feed handler API* on page 287

DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_handler_get_info()**

This function returns the handler information for the feed handler.

The handler information is the value of the HANDLERNXX_INFO parameter in the configuration file, where XX is the handler ID that is passed to the function.

**Syntax**

```c
char * rtl_handler_get_info(
    int p_handler_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **p_handler_id**
  - The ID number of the handler.

**Return values**

If **rtl_handler_get_info()** is successful, it returns the information.

If **rtl_handler_get_info()** fails, it returns a zero-length string.

If the feed handler specified by the handler ID is not configured, this function returns a zero-length string. You can use the **rtl_handler_is_configured()** function to determine if a feed handler for a particular handler ID is configured.

**Note:** The HANDLERNXX_INFO config parameter is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

- [“Feed handler API” on page 287](#)

  DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_handler_get_name()**

This function returns the name for the feed handler.

The handler name is the value of the HANDLERXX_NAME parameter in the configuration file, where XX is the handler ID that is passed to the function.

### Syntax

```c
char * rtl_handler_get_name(
    int p_handler_id);
```

### Arguments

**p_handler_id**

The ID number of the handler.

### Return values

If `rtl_handler_get_name()` is successful, it returns the handler name.

If `rtl_handler_get_name()` fails, it returns a zero-length string.

If the feed handler specified by the handler ID is not configured, this function returns a zero-length string. You can use the `rtl_handler_is_configured()` function to determine if a feed handler for a particular handler ID is configured.

No No Yes No No

### Related concepts

[“Feed handler API” on page 287](#)

DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_handler_get_user_data()**

This function returns the handler user data for the feed handler. The handler user data is the memory buffer allocated by the feed handler and is returned to DSE from the rtl_handler_init() call in the feed handler implementation.

The contents and use of that area are determined by the feed handler implementation.

**Syntax**

```c
void * rtl_handler_get_user_data(
    int p_handler_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

**p_handler_id**

The ID number of the handler

**Return values**

If rtl_handler_get_user_data() is successful, it returns a pointer to the handler user data area.

If rtl_handler_get_user_data() fails or no user data has been allocated, it returns null.

**Related concepts**

- "Feed handler API" on page 287
- DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_handler_is_configured()**

This function enables you to determine if a particular feed handler is defined in the configuration file. You can use this function to check the validity of a feed handler ID before calling other handler accessor functions, such as `rtl_handler_get_info()`.

**Syntax**

```c
int rtl_handler_is_configured(
    int p_handler_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`p_handler_id`

The ID number of the handler

**Return values**

The `rtl_handler_is_configured()` function returns 1 if the feed handler for the specified ID is configured. Otherwise, it returns 0.

**Related concepts**

- "Feed handler API" on page 287
- DSE provides API functions for feed handlers to retrieve information about the feed.
**rtl_handler_log()**

This function provides a means for messages to be displayed in the DSE message log. The message file that a handler uses is defined by the value of the HANDLERXX_MSG_PATH NAME parameter in the configuration file.

**Syntax**

```c
void rtl_handler_log(
    int p_handler_id,
    int p_errno,
    ...);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

**p_handler_id**

The ID number of the handler.

**p_errno**

The error number from the message file that should be printed.

**...**

A printf-style variable-length list of arguments corresponding to parameters in the log message.

**Return values**

The rtl_handler_log() function has no return values.
**rtl_hasys_get_info()**

This function returns the user information that is supplied in the HA_INFO configuration parameter.

**Syntax**

```
char *rtl_hasys_get_info();
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Return values**

This function returns a pointer to the string that is supplied in the HA_INFO configuration parameter. If the HA_INFO parameter is not defined, this function returns null.

**Related concepts**

- “High availability information accessor API” on page 288
- DSE provides API functions for the high availability communication library to use. The purpose of these API functions is to retrieve information about the high availability configuration.
**rtl_hasys_get_user_data()**

This function returns a pointer to the high availability user data area that is allocated by the rtl_ha_init() function.

**Syntax**

```c
void *rtl_hasys_get_ha_user_data();
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Return values**

This function returns a pointer to the high availability user data area that is allocated by the rtl_ha_init() function.

If rtl_hasys_get_user_data() fails or no user data has been allocated, it returns null.

**Related concepts**

- "High availability information accessor API" on page 288

DSE provides API functions for the high availability communication library to use. The purpose of these API functions is to retrieve information about the high availability configuration.
**rtl_localtime_to_rtltime()**

This function generates an rtl_datetime type timestamp for the current local time.

The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
void rtl_localtime_to_rtltime(
    rtl_datetime *rtl_time);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

rtl_time

The current local time as an rtl_datetime type (output parameter).

**Return values**

The rtl_localtime_to_rtltime function has no return values.

**Related concepts**

"Time API" on page 291

DSE provides API functions for managing timestamps.
**rtl_minus_seconds()**

This function subtracts a number of seconds from an rtl_datetime type timestamp.

The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```plaintext
void rtl_minus_seconds(
    rtl_datetime *rtl_time,
    long inc,
    rtl_datetime *ret);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_time**
  The original timestamp as an rtl_datetime type (input parameter).

- **inc**
  The number of seconds to subtract from the original timestamp (input parameter).

- **ret**
  The resulting timestamp with subtracted seconds as an rtl_datetime type (output parameter).

**Return values**

The rtl_minus_seconds() function has no return values.

**Related concepts**

- "Time API" on page 291

DSE provides API functions for managing timestamps.
**rtl_put_metadata()**

This function updates one field of a symbol’s metadata in shared memory.

**Syntax**

```c
int rtl_put_metadata(
    rtldesc *rdesc,
    char *name,
    char *field_name,
    void *data,
    int data_len,
    char isnull)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **rdesc**  A valid rtldesc that is returned by rtlattach or rtlreattach.
- **name**   Name of the symbol. name can identify only a single symbol.
- **field_name**  Name of the field in the metadata that you want to update. field_name must match the name of a field specified by the RECORDXX_SCHEMA or RECORDXX_METADATA_SCHEMA definition.
- **data**    A pointer to the field data to write.
- **data_len**  Length of the data referred to in data.
- **is_null**  If the field needs to be null, set to 1.

**Return values**

If rtl_put_metadata() is successful, it returns RTLAPI_SUCCESS.

If rtl_put_metadata() fails, it returns one of the following return codes:

**Table 46. Return Codes**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_BAD_DESC</td>
<td>rdesc is null or points to an invalid rtldesc structure.</td>
</tr>
<tr>
<td>RTLAPI_NO_SUCH_NAME</td>
<td>name is not found in the store.</td>
</tr>
<tr>
<td>RTLAPI_BAD_FAILED</td>
<td>field_name is a null pointer or contains an unknown field name.</td>
</tr>
<tr>
<td>RTLAPI_MALLOC_FAILED</td>
<td>Either DSE or the client API could not allocate space.</td>
</tr>
<tr>
<td>RTLAPI_NOT_CONNECTED</td>
<td>Not connected to a store.</td>
</tr>
<tr>
<td>RTLAPI_IN_SCAN</td>
<td>Already in scan.</td>
</tr>
<tr>
<td>RTLAPI_BAD_NAME</td>
<td>name is null.</td>
</tr>
<tr>
<td>RTLAPI_BAD_METADATA</td>
<td>data is null.</td>
</tr>
<tr>
<td>RTLAPI_BAD_METADATALEN</td>
<td>data_len is less than or equal to 0.</td>
</tr>
</tbody>
</table>
### Table 46. Return Codes (continued)

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Communication error with DSE.</td>
</tr>
<tr>
<td>RTLAPI_META_SET_ERROR</td>
<td>Could not write metadata to store.</td>
</tr>
<tr>
<td>RTLAPI_NULL_VIOLATION</td>
<td>The is_null argument is set and the specified field has a NOT NULL constraint.</td>
</tr>
</tbody>
</table>

**Related concepts**

[C client API](#) on page 285

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_reattach()**

This function switches to a new data store once a connection has been established by the rtl_attach() function.

The new store must be on the same instance of DSE to which you originally attached. The rtl_reattach() function uses the group authorization level that is determined by the rtl_attach() function.

Client applications can use the rtl_reattach() function to get statistics for stores that use a simple schema, if the storagesys_name and store_name argument values are null.

**Syntax**

```c
int rtl_reattach(
    rtldesc *rdesc,
    char *storagesys_name,
    char *store_name)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **rdesc** A valid rtldesc that is returned by the rtl_attach() function or the rtl_reattach() function.

- **storagesys_name**
  
  Name of a storage system that is defined in the rtlconfig file.

  If storagesys_name is null, DSE will detach the client from the storage system to which it is currently attached.

  This value is null for stores that use a simple schema.

- **store_name**
  
  Name of a data store in the specified storage system.

  If store_name is null, DSE detaches the client from the store to which it is currently attached.

  This value is null for stores that use a simple schema.

**Return values**

If rtl_reattach() is successful, it returns RTLAPI_SUCCESS.

If rtl_reattach() fails, it returns one of the following return codes:

**Table 47. Return Codes**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_ATTACH_REFUSED</td>
<td>This function does not support stores that use a simple schema.</td>
</tr>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Connection to DSE failed or dropped.</td>
</tr>
<tr>
<td>RTLAPI_NOT_AUTH</td>
<td>Authority of the store is greater than the authority of the group.</td>
</tr>
<tr>
<td>RTLAPI_NO_SUCH_STORESYS</td>
<td>Storage system does not exist.</td>
</tr>
</tbody>
</table>
Table 47. Return Codes (continued)

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_NO_SUCH_STORE</td>
<td><em>store_name</em> does not exist in the storage system space.</td>
</tr>
<tr>
<td>RTLAPI_WRONG_VERSION</td>
<td>Client library version is out of sync with DSE.</td>
</tr>
<tr>
<td>RTLAPI_BADSTORESYSNAME</td>
<td><em>storagesys_name</em> is longer than 256 characters.</td>
</tr>
<tr>
<td>RTLAPI_BADSTORENAME</td>
<td><em>store_name</em> is longer than 256 characters.</td>
</tr>
<tr>
<td>RTLAPI_MALLOC_FAILED</td>
<td>Either DSE or the client API could not allocate space.</td>
</tr>
<tr>
<td>RTLAPI_IN_SCAN</td>
<td>rtl_attach was called at the incorrect time.</td>
</tr>
<tr>
<td>RTLAPI_BAD_DESC</td>
<td><em>rdesc</em> is null or points to an invalid rtldesc structure.</td>
</tr>
<tr>
<td>RTLAPI_STORESYS_LOAD_ERR</td>
<td>Could not load shared library implementing the storage system.</td>
</tr>
<tr>
<td>RTLAPI_STORESYS_OPEN_ERR</td>
<td>Could not open the storage system.</td>
</tr>
<tr>
<td>RTLAPI_STORE_OPEN_ERR</td>
<td>Could not open the store.</td>
</tr>
<tr>
<td>RTLAPI_ATTACH_REFUSED</td>
<td>Reattach not allowed to:</td>
</tr>
<tr>
<td></td>
<td>• A standby DSE instance.</td>
</tr>
<tr>
<td></td>
<td>• A store using a simple schema.</td>
</tr>
</tbody>
</table>

**Related concepts**

[C client API” on page 285](#)

DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_rtljavadate_to_rtltime()**

This function converts a timestamp from an rtl_java_date type to an rtl_datetime type.

The rtl_java_date is a DSE 64-bit date time representation with millisecond precision based on the Java.Util.Date representation as a long type. The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
void rtl_rtljavadate_to_rtltime(
    rtl_java_date rtl_jd,
    rtl_datetime *rtl_time);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_jd**  The timestamp as an rtl_java_date type (input parameter).
- **rtl_time**  The timestamp as an rtl_datetime type (output parameter).

**Return values**

The rtl_rtljavadate_to_rtltime() function has no return values.

**Related concepts**

-Time API on page 291-

DSE provides API functions for managing timestamps.
rtl_rtltime_cmp()

This function compares two rtl_datetime timestamps.

Syntax

```c
int rtl_rtltime_cmp(
    rtl_datetime *rtl_time1,
    rtl_datetime *rtl_time2);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Arguments

rtl_time1

The first timestamp as an rtl_datetime type (input parameter).

rtl_time2

The second timestamp as an rtl_datetime type (input parameter).

Return values

Similar to the C functions memcmp and strcmp:

- If rtl_time1 is greater than rtl_time2, this function returns an integer that is greater than 0.
- If rtl_time1 is equal to rtl_time2, this function returns 0.
- If rtl_time1 is less than rtl_time2, this function returns an integer that is less than 0.

Related concepts

"Time API" on page 291

DSE provides API functions for managing timestamps.
**rtl_rtltime_cpy()**

This function copies a source rtl_datetime timestamp to a target rtl_datetime timestamp.

**Syntax**

```c
void rtl_rtltime_cpy(
    rtl_datetime *rtl_time1,
    rtl_datetime *rtl_time2);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_time1**
  - The target timestamp as an rtl_datetime type (output parameter).

- **rtl_time2**
  - The source timestamp as an rtl_datetime type (input parameter).

**Return values**

The rtl_rtltime_cpy() function has no return values.

**Related concepts**

- "Time API" on page 291
- DSE provides API functions for managing timestamps.
**rtl_rtltime_date_to_string()**

This function converts the date part of an rtl_datetime type timestamp to a string representation in ISO 8601 format YYYY-MM-DD.

The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
void rtl_rtltime_date_to_string(
    rtl_datetime *rtl_time,
    char       *buf);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

**rtl_time**

The timestamp as an rtl_datetime type (input parameter).

**buf**

The date part of the timestamp as a character string (output parameter).

**Note:** The memory pointed to by the buf parameter should be allocated by the caller and be of sufficient size to accommodate the output string.

**Return values**

The rtl_rtltime_date_to_string() function has no return values.

**Related concepts**

- "Time API" on page 291

DSE provides API functions for managing timestamps.
rtl rtltime_time_to_string()

This function converts the time part of an rtl_datetime type timestamp to a string representation in ISO 8601 format HH:MM:SS.

The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

Syntax

```c
void rtl rtltime_time_to_string(
    rtl_datetime *rtl_time,
    char      *buf);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Arguments

rtl_time

The timestamp as an rtl_datetime type (input parameter).

buf

The date part of the timestamp as a character string (output parameter).

Note: The memory pointed to by the buf parameter should be allocated by the caller and be of sufficient size to accommodate the output string.

Return values

The rtl_rtltime_time_to_string() function has no return values.

Related concepts

"Time API" on page 291

DSE provides API functions for managing timestamps.
**rtl_rtltime_to_rtljavadate()**

This function converts a timestamp from an rtl_datetime type to an rtl_java_date type.

The rtl_java_date is a DSE 64-bit date time representation with millisecond precision based on the Java.Util.Date representation as a long type. The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
rtl_java_date rtl_rtltime_to_rtljavadate(
    rtl_datetime *rtl_time);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

rtl_time

The timestamp as an rtl_datetime type (input parameter).

**Return values**

The timestamp as an rtl_java_date type.

**Related concepts**

"Time API" on page 291

DSE provides API functions for managing timestamps.
**rtl_rtltime_to_string()**

This function converts a timestamp from an rtl_datetime type to a timestamp string representation in ISO 8601 format YYYY-MM-DD HH:MM:SS.FFFF.

The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
void rtl_rtltime_to_string(
    rtl_datetime *rtl_time,
    char     *buf);
```

**Valid in client application?** | **Valid in storage system?** | **Valid in feed handler?** | **Valid in HA library?** | **Valid in transport?**
---|---|---|---|---
Yes | Yes | Yes | Yes | Yes

**Arguments**

**rtl_time**  
The timestamp as an rtl_datetime type (input parameter).

**buf**  
The timestamp as a character string (output parameter).

**Note:** The memory pointed to by the buf parameter should be allocated by the caller and be of sufficient size to accommodate the output string.

**Return values**

The rtl_rtltime_to_string() function has no return values.

**Related concepts**

[“Time API” on page 291](#)

DSE provides API functions for managing timestamps.
rtl_rtltime_to_timet()

This function converts a timestamp from an rtl_datetime type to a time_t type.

The time_t type is a C date time representation with year to second precision defined in time.h. The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

Syntax

time_t rtl_rtltime_to_timet(
    rtl_datetime *rtl_time);

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Arguments

rtl_time

The timestamp as an rtl_datetime type (input parameter).

Return values

The timestamp as a time_t type.

Related concepts

"Time API" on page 291
DSE provides API functions for managing timestamps.
**rtl_rtltime_to_tm()**

This function converts a timestamp from an rtl_datetime type to a tm struct type.

The tm struct is a C date time representation with year to second precision defined in time.h. The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
struct tm rtl_rtltime_to_tm(
   rtl_datetime *rtl_time);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

**rtl_time**

The timestamp as an rtl_datetime type (input parameter).

**Return values**

The timestamp as a tm struct type.

**Related concepts**

"Time API" on page 291

DSE provides API functions for managing timestamps.
rtl_set_meta_defaults()

This function sets a provided metadata area and corresponding nulls array to the default values specified by the RECORDXX_META_SCHEMA parameter in the configuration file.

If a field has a default value, it is copied to the appropriate place in the metadata area. The nulls array is set to indicate that a field is:
- Null for all fields without a default value.
- Not null for all fields with a default value.

**Syntax**

```c
int rtl_set_meta_defaults(
    int rectype,
    void *sechdr,
    char *nulls);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **rectype**
  The numeric ID of the record.
- **sechdr**
  Pointer to a metadata area allocated by the caller to be populated with defaults (output parameter).
- **nulls**
  Pointer to a nulls array allocated by the caller to be populated with defaults (output parameter).

**Return values**

If the rtl_set_meta_defaults() function is successful, it returns RTL_SUCCESS.

If the rtl_set_meta_defaults() function fails, it returns RTL_FAILURE.

**Related concepts**

- "Record type API" on page 289
DSE provides API functions for retrieving information about record types and setting default values for record types.
**rtl_set_rtltime_fields()**

This function sets the date and time components of an rtl_datetime type.

**Syntax**

```c
void rtl_set_rtltime_fields(
    rtl_datetime *rtl_time,
    int year,
    int month,
    int day,
    int hour,
    int minute,
    int second,
    int microsecond);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **rtl_time**
  
  The timestamp as an rtl_datetime type (output parameter).

- **year**
  
  The year component of the rtl_datetime timestamp (input parameter).

- **month**
  
  The month component of the rtl_datetime timestamp (input parameter).

- **day**
  
  The day component of the rtl_datetime timestamp (input parameter).

- **hour**
  
  The hour component of the rtl_datetime timestamp (input parameter).

- **minute**
  
  The minute component of the rtl_datetime timestamp (input parameter).

- **second**
  
  The second component of the rtl_datetime timestamp (input parameter).

- **microsecond**
  
  The microsecond component of the rtl_datetime timestamp (input parameter).

**Return values**

The rtl_set_rtltime_fields() function has no return values.

**Related concepts**

- "Time API" on page 291

DSE provides API functions for managing timestamps.
**rtl_set_tick_defaults()**

This function sets a provided tick and corresponding nulls array to the default values specified by the RECORDXX_SCHEMA parameter in the configuration file.

If a field has a default value it is copied to the appropriate place in the tick. The nulls array is set to indicate that a field is:
- Null for all fields without a default value.
- Not null for all fields with a default value.

**Syntax**

```c
int rtl_set_tick_defaults(
    int rectype,
    void *tick,
    char *nulls);
```

**Arguments**

- **rectype**
  The numeric ID of the record.

- **tick**
  Pointer to a tick allocated by the caller to be populated with defaults (output parameter).

- **nulls**
  Pointer to a nulls array allocated by the caller to be populated with defaults (output parameter).

**Return values**

If the rtl_set_tick_defaults() function is successful, it returns RTL_SUCCESS.

If the rtl_set_tick_defaults() function fails, it returns RTL_FAILURE.

**Related concepts**

"Record type API" on page 289

DSE provides API functions for retrieving information about record types and setting default values for record types.
**rtl_state_change_register()**

This function registers a function to be called whenever the state of the DSE instance changes.

The arguments of this function have the same meaning as the output arguments of the rtl_state_get() function.

**Syntax**

```c
void rtl_state_change_register(void (*state_change)(rtl_state_t, float));
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*state_change* (in)

The callback procedure that receives the state and the value of the pctn_insync argument when the state of the DSE instance changes.

**Return values**

The rtl_state_change_register() function has no return values.

**Related concepts**

“High availability state API” on page 288

The high availability state API provides functions that custom code can use in a DSE instance to manage the high availability state of DSE.
**rtl_state_get()**

This function returns the current state of the DSE instance. This function performs the same task as the rtlstat -h command.

**Syntax**

```c
rtl_state_t rtl_state_get(float *pcnt_insync);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`pcnt_insync (out)`

If this argument is provided, it returns an indicator of how much synchronization has been done by the standby DSE instance. Its values range from 0.0 to 1.0:

- 0.0  The DSE instance is in STARTED state.
- 0.33 Synchronization of symbol names has started.
- 0.66 Synchronization of metadata has started.
- 1.0  The DSE instance is in STANDBY state.

**Return values**

This function returns the current state of the DSE instance. The possible states are:

- `RTL_STATE_ACTIVE`
- `RTL_STATE_STARTED`
- `RTL_STATE_SYNCHRONIZING`
- `RTL_STATE_STANDBY`
- `RTL_STATE_INACTIVE`

**Related concepts**

"High availability state API" on page 288

The high availability state API provides functions that custom code can use in a DSE instance to manage the high availability state of DSE.
**rtl_state_get_info()**

This function returns the user information that is supplied in the HA_INFO configuration parameter.

**Syntax**

```c
char *rtl_state_get_info();
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Return values**

If the rtl_state_get_info() function is successful, it returns a pointer to the string that is supplied in the HA_INFO configuration parameter.

If the rtl_state_get_info() function fails, it returns null.

**Note:** The HA_INFO config parameter is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

[“High availability state API” on page 288](#)

The high availability state API provides functions that custom code can use in a DSE instance to manage the high availability state of DSE.
**rtl_state_set()**

This function changes the state of a DSE instance.

**Syntax**

```c
int rtl_state_set(
    rtl_state_t state,
    int force);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

**state** (in)

The new state. The possible states are:

- RTL_STATE_ACTIVE
- RTL_STATE_STARTED
- RTL_STATE_SYNCHRONIZING
- RTL_STATE_STANDBY
- RTL_STATE_INACTIVE

The RTL_STATE_ACTIVE state forces a DSE instance that is in STARTED or STANDBY state to go into ACTIVE state. This function performs the same task as the rtlmode -M I command.

The RTL_STATE_INACTIVE state forces a DSE instance that is in any state to go into INACTIVE state. This function performs the same task as the rtlmode -k command.

**force** (in)

Shut down DSE immediately (this occurs only if the state is RTL_STATE_INACTIVE).

If the force flag is set, this function shuts down DSE immediately. This function performs the same task as the rtlmode -x command. The force flag is available only if the state is being set to INACTIVE.

**Return values**

If the rtl_state_set() function is successful, it returns RTL_SUCCESS.

If the rtl_state_set() function fails, it returns RTL_FAILURE.

**Related concepts**

"High availability state API" on page 288

The high availability state API provides functions that custom code can use in a DSE instance to manage the high availability state of DSE.
rtl_stats()

This function retrieves statistical information from DSE. Use this information to monitor DSE activities.

Syntax

```c
int rtl_stats(
    rtldesc *rdesc,
    int type,
    stat_response **ret RESP
)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

**rdesc**  A valid `rtldesc` that is returned by `rtl_attach` or `rtl_reattach`.

**type**  Type of statistics that you want to be returned.

The possible values are:

**RTLSTAT_DISCARDS**  Returns statistics about bad data that is discarded from shared memory to a discard table.

**RTLSTAT_FLUSHES**  Returns various fluser statistics.

**RTLSTAT_INSERTS**  Returns various new-symbol insert statistics.

**RTLSTAT_LOCKS**  Returns spinlock statistics.

**RTLSTAT_METAWRITES**  Returns statistics about metadata writes.

**RTLSTAT_PERF**  Returns general statistics.

**RTLSTAT_READ_RATE**  Returns the number of messages that have been read since system startup.

**RTLSTAT_READS**  Returns various feed statistics.

**RTLSTAT_SHM**  Returns statistics about shared memory usage.

**RTLSTATSTORES**  Returns ticks that have been received and flushed for each store.

**RTLSTAT_SUMMARY**  Returns statistics summary.

**ret RESP**  Points to a union of all possible statistics.

The actual statistics that are returned are determined by `type`. See `rtl_client.h` for the definition of each of the statistics structures.
Return values

If rtl_stats() is successful, it returns RTLAPI_SUCCESS.

If rtl_stats() fails, it returns one of the following return codes:

Table 48. Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLAPI_BAD_DESC</td>
<td>rdesc is null or points to an invalid rtldesc structure.</td>
</tr>
<tr>
<td>RTLAPI_MALLOC_FAILED</td>
<td>Either DSE or the client API could not allocate space.</td>
</tr>
<tr>
<td>RTLAPI_BAD_STAT_RESP</td>
<td>ret_resp is null.</td>
</tr>
<tr>
<td>RTLAPI_IO_FAILED</td>
<td>Communication error with DSE.</td>
</tr>
<tr>
<td>RTLAPI_BAD_STAT_TYPE</td>
<td>type is not a valid statistics type.</td>
</tr>
</tbody>
</table>

Related concepts

“C client API” on page 285
DSE provides API functions for writing C client applications. These API functions serve two main purposes: to manage connections to DSE, and to retrieve data from data stores through DSE.
**rtl_store_get_flush_batch_size_from_desc()**

This function returns the tick flush batch size from the store descriptor.

The batch size is the value of the STOREXX_FLUSH_BATCH parameter in the configuration file.

**Syntax**

```c
int rtl_store_get_flush_batch_size_from_desc(void *store_desc);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_desc*

The descriptor of the store.

**Return values**

If `rtl_store_get_flush_batch_size_from_desc()` is successful, it returns the tick flush batch size.

If `rtl_store_get_flush_batch_size_from_desc()` fails, it returns -1.

**Related concepts**

“Store API” on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
rtl_store_get_flush_batch_size_from_id()

This function returns the tick flush batch size from the store ID.

The batch size is the value of the STOREXX_FLUSH_BATCH parameter in the configuration file.

**Syntax**

```c
int rtl_store_get_flush_batch_size_from_id(int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_id*

The ID number of the store.

**Return values**

If `rtl_store_get_flush_batch_size_from_id()` is successful, it returns the tick flush batch size.

If the store specified by the store ID is not configured, this function returns -1. You can use the `rtl_store_is_configured()` function to determine if a store for a particular store ID is configured.

**Related concepts**

"Store API" on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_id_from_desc()**

This function returns the store ID from the store descriptor.

This ID is the XX part of the store entry (for example, STOREXX_NAME) in the configuration file for the given store.

**Syntax**

```c
int rtl_store_get_id_from_desc(
    void *store);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store*   The descriptor of the store.

**Return values**

If `rtl_store_get_id_from_desc()` is successful, it returns the store ID (0-99).

If `rtl_store_get_id_from_desc()` fails, it returns -1.

**Related concepts**

["Store API" on page 289](#)

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
rtl_store_get_id_from_name()

This function returns the ID of the given store.

Syntax

```c
int rtl_store_get_id_from_name(
    char *store_name);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

store_name

The name of the store of the store.

Return values

If rtl_store_get_id_from_name() is successful, it returns the store ID.

If rtl_store_get_id_from_name() fails, it returns -1.

Related concepts

[“Store API” on page 289]

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_info_from_desc()**

This function returns the store info from the store descriptor.

This information is the STOREXX_INFO part of the store entry in the configuration file for the given store.

**Syntax**

```c
char *rtl_store_get_info_from_desc(
    void *store);
```

<table>
<thead>
<tr>
<th>Valid in client application</th>
<th>Valid in storage system</th>
<th>Valid in feed handler</th>
<th>Valid in HA library</th>
<th>Valid in transport</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The descriptor of the store.

**Return values**

If `rtl_store_get_info_from_desc()` is successful, it returns the store info.

If `rtl_store_get_info_from_desc()` fails, it returns a zero-length string.

**Note:** The information from the STOREXX_INFO part of the store entry is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

[“Store API” on page 289](#)

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_info_from_id()**

This function returns the store info from the store ID.

This information is the STOREXX_INFO part of the store entry in the configuration file for the given store.

**Syntax**

```c
char *rtl_store_get_info_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_id*

The ID of the storage system.

**Return values**

If `rtl_store_get_info_from_id()` is successful, it returns the store information.

If `rtl_store_get_info_from_id()` fails, it returns a zero-length string.

If the store specified by the store ID is not configured, this function returns a zero-length string. You can use the `rtl_store_is_configured()` function to determine if a store for a particular store ID is configured.

**Note:** The information from the STOREXX_INFO part of the store entry is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

["Store API" on page 289](#)

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
### rtl_store_get_meta_batch_size_from_desc()

This function returns the metadata persistence batch size from the store descriptor.

The batch size is the value of the STOREXX_META_BATCH parameter in the configuration file.

#### Syntax

```c
int rtl_store_get_meta_batch_size_from_desc(void *store_desc);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

#### Arguments

- **store_desc**
  - The descriptor of the store.

#### Return values

If `rtl_store_get_meta_batch_size_from_desc()` is successful, it returns the metadata persistence batch size.

If `rtl_store_get_meta_batch_size_from_desc()` fails, it returns -1.

#### Related concepts

- ["Store API" on page 289](#)
  - DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
rtl_store_get_meta_batch_size_from_id()

This function returns the metadata persistence batch size from the store ID.

The batch size is the value of the STOREXX_META_BATCH parameter in the configuration file.

Syntax

```
int rtl_store_get_meta_batch_size_from_id(int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

store_id

The ID number of the store.

Return values

If rtl_store_get_meta_batch_size_from_id() is successful, it returns the metadata persistence batch size.

If the store specified by the store ID is not configured, this function returns -1. You can use the rtl_store_is_configured() function to determine if a store for a particular store ID is configured.

Related concepts

“Store API” on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_meta_mode_from_desc()**

This function returns the metadata persistence mode from the store descriptor.

The mode is the value of the STOREXX_META_MODE parameter in the configuration file.

**Syntax**

```c
rtl_store_meta_mode_t rtl_store_get_meta_mode_from_desc(void *store_desc);
```

---

**Valid in client application?** | **Valid in storage system?** | **Valid in feed handler?** | **Valid in HA library?** | **Valid in transport?**
---|---|---|---|---
No | Yes | No | No | No

---

**Arguments**

*store_desc*

The descriptor of the store.

**Return values**

If `rtl_store_get_meta_mode_from_desc()` is successful, it returns the metadata persistence mode. The mode can be `RTL_STORE_META_MODE_DISABLED`, `RTL_STORE_META_MODE_MANUAL`, or `RTL_STORE_META_MODE_AUTO`.

If `rtl_store_get_meta_mode_from_desc()` fails, it returns -1.

**Related concepts**

["Store API" on page 289]

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_meta_mode_from_id()**

This function returns the metadata persistence mode from the store ID.

The mode is the value of the STOREXX_META_MODE parameter in the configuration file.

**Syntax**

```c
rtl_store_meta_mode_t rtl_store_get_meta_mode_from_id(int store_id);
```

**Arguments**

- **store_id**
  The ID number of the store.

**Return values**

If `rtl_store_get_meta_mode_from_id()` is successful, it returns the metadata persistence mode. The mode can be `RTL_STORE_META_MODE_DISABLED`, `RTL_STORE_META_MODE_MANUAL`, or `RTL_STORE_META_MODE_AUTO`.

If the store specified by the store ID is not configured, this function returns -1. You can use the `rtl_store_is_configured()` function to determine if a store for a particular store ID is configured.

**Related concepts**

- ["Store API" on page 289](#)
- DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_meta_period_from_desc()**

This function returns the metadata persistence period from the store descriptor.

The metadata persistence period is the value of the STOREXX_META_PERIOD parameter in the configuration file.

**Syntax**

```c
int rtl_store_get_meta_period_from_desc(void *store_desc);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store_desc

The descriptor of the store.

**Return values**

If `rtl_store_get_meta_period_from_desc()` is successful, it returns the metadata persistence period.

If `rtl_store_get_meta_period_from_desc()` fails, it returns -1.

**Related concepts**

"Store API" on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
rtl_store_get_meta_period_from_id()

This function returns the metadata persistence period from the store ID.

The period is the value of the STOREXX_META_PERIOD parameter in the configuration file.

Syntax

```c
int rtl_store_get_meta_period_from_id(int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

store_id

The ID number of the store.

Return values

If rtl_store_get_meta_period_from_id() is successful, it returns the metadata persistence period.

If the store specified by the store ID is not configured, this function returns -1. You can use the rtl_store_is_configured() function to determine if a store for a particular store ID is configured.

Related concepts

“Store API” on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_name_from_desc()**

This function returns the store name from the store descriptor.

This name is the STOREXX_NAME part of the store entry in the configuration file for the given store.

**Syntax**

```c
char *rtl_storesys_get_name_from_desc(
    void *store);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The descriptor of the store.

**Return values**

If rtl_store_get_name_from_desc() is successful, it returns the store name.

If rtl_store_get_name_from_desc() fails, it returns a zero-length string.

**Related concepts**

- "Store API" on page 289
- DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_name_from_id()**

This function returns the store name from the store ID.

This name is the STOREXX_NAME part of the store entry in the configuration file for the given store.

**Syntax**

```c
char *rtl_store_get_name_from_id(
    int `store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_id*

The ID number of the store.

**Return values**

If rtl_store_get_name_from_id() is successful, it returns the store name.

If rtl_store_get_name_from_id() fails, it returns a zero-length string.

If the store specified by the store ID is not configured, this function returns a zero-length string. You can use the rtl_store_is_configured() function to determine if a store is configured for a particular store ID.

**Related concepts**

“Store API” on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_num_flushers_from_desc()**

This function returns the number of flushers the store is using from the store descriptor.

**Syntax**

```c
int rtl_store_get_num_flushers_from_desc(
    void *store);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The descriptor of the store.

**Return values**

If `rtl_store_get_num_flushers_from_desc()` is successful, it returns the number of flushers.

If `rtl_store_get_num_flushers_from_desc()` fails, it returns -1.

**Related concepts**

- "Store API" on page 289
- DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_num_flushers_from_id()**

This function returns the number of flushers that the store is using from the store ID.

**Syntax**

```c
int rtl_store_get_num_flushers_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_id*

The ID number of the store.

**Return values**

If **rtl_store_get_num_flushers_from_id()** is successful, it returns the number of flushers.

If the store specified by the store ID is not configured, this function returns -1. You can use the **rtl_store_is_configured()** function to determine if a store for a particular store ID is configured.

**Related concepts**

"Store API" on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_partition_mode_from_desc()**

This function returns the partition mode for the store from the store descriptor.

**Syntax**

```c
int rtl_store_get_partition_mode_from_desc(void *store_desc)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **store_desc**
  - The descriptor of the store.

**Return values**

If `rtl_store_get_partition_mode_from_desc()` is successful, it returns the store partition mode. The store partition modes are:

**RTL_TICK_PARTITION**
- The store is partitioning data on a per-tick basis.

**RTL_SYMBOL_PARTITION**
- The store is partitioning data on a per-symbol basis.

**RTL_NO_PARTITION**
- The store is not partitioning data.

If this function is called before the partition mode is set by calling the `rtl_store_get_partition_mode()` function, then this function returns `RTL_UNINITIALIZED_PARTITION_MODE`.

If `rtl_store_get_partition_mode_from_desc()` fails, it returns `RTL_FAILURE`.

**Related concepts**

> "Store API" on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_partition_mode_from_id()**

This function returns the partition mode for the store from the store ID.

**Syntax**

```c
int rtl_store_get_partition_mode_from_id (int store_id)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store_id

The ID number of the store.

**Return values**

If `rtl_store_get_partition_mode_from_id()` is successful, it returns the store partition mode. The store partition modes are:

**RTL_TICK_PARTITION**

The store is partitioning data on a per-tick basis.

**RTL_SYMBOL_PARTITION**

The store is partitioning data on a per-symbol basis.

**RTL_NO_PARTITION**

The store is not partitioning data.

If this function is called before the partition mode is set by calling the `rtl_store_get_partition_mode()` function, then this function returns `RTL_UNINITIALIZED_PARTITION_MODE`.

If the store that is specified by the store ID is not configured, this function returns `RTL_FAILURE`. Use the `rtl_store_is_configured()` function to determine if a store for a particular store ID is configured.

**Related concepts**

- “Store API” on page 289
  
  DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_rectype_from_desc()**

This function returns the store rectype number from the store descriptor.

**Syntax**

```c
int rtl_store_get_rectype_from_desc(
    void *store);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The descriptor of the store.

**Return values**

If `rtl_store_get_rectype_from_desc()` is successful, it returns the rectype ID.

If `rtl_store_get_rectype_from_desc()` fails, it returns -1.

**Related concepts**

- "Store API" on page 289

DSE provides this API for accessing information about a store from either the store descriptor or a store ID.
**rtl_store_get_rectype_from_id()**

This function returns the store rectype number that is associated with the store ID.

**Syntax**

```c
int rtl_store_get_rectype_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store_id

The ID number of the store.

**Return values**

If `rtl_store_get_rectype_from_id()` is successful, it returns the rectype ID.

If the store specified by the store ID is not configured, this function returns -1. You can use the `rtl_store_is_configured()` function to determine if a store for a particular store ID is configured.
rtl_store_get_schema_name_from_desc()

This function returns the store schema from the store descriptor.

The schema comes from the STOREXX_SCHEMA part of the store entry in the configuration file for the given store.

**Syntax**

```c
char * rtl_store_get_schema_name_from_desc(
    void *store_desc);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`store_desc`  
The descriptor of the store.

**Return values**

If `rtl_store_get_schema_name_from_desc()` is successful, it returns the store schema.

If `rtl_store_get_schema_name_from_desc()` fails, it returns a zero-length string.
**rtl_store_get_schema_name_from_id()**

This function returns the store schema from the store ID.

The schema comes from the STOREXX_SCHEMA part of the store entry in the configuration file for the given store.

**Syntax**

```c
char *rtl_store_get_schema_name_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store_id

The ID number of the store.

**Return values**

If `rtl_store_get_schema_name_from_id()` is successful, it returns the store schema.

If `rtl_store_get_schema_name_from_id()` fails, it returns a zero-length string.
rtl_store_get_schema_type_from_desc()

This function returns the store schema type from the store descriptor.

**Syntax**

```c
rtl_store_schema_type_t rtl_store_get_schema_type_from_desc(
    void *store_desc);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_desc*

The store descriptor.

**Return values**

If `rtl_store_get_schema_type_from_desc()` is successful, it returns the store schema type. The store schema type can be:

- `RTL_STORE_MASTER_DETAIL_SCHEMA`
- `RTL_STORE_SIMPLE_SCHEMA`

If `rtl_store_get_schema_type_from_desc()` fails, it returns -1.
**rtl_store_get_schema_type_from_id()**

This function returns the store schema type from the store ID.

**Syntax**

```c
rtl_store_schema_type_t rtl_store_get_schema_type_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_id*

The ID number of the store.

**Return values**

If `rtl_store_get_schema_type_from_id()` is successful, it returns the store schema type. The store schema type can be:

- `RTL_STORE_MASTER_DETAIL_SCHEMA`
- `RTL_STORE_SIMPLE_SCHEMA`

If `rtl_store_get_schema_type_from_id()` fails, it returns -1. You can use the `rtl_store_is_configured()` function to determine whether a store with a particular store ID is configured.
**rtl_store_get_storesys_id_from_desc()**

This function returns the storage system number from the store descriptor.

**Syntax**

```c
int rtl_store_get_storesys_id_from_desc(
    void  *store);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The descriptor of the store.

**Return values**

If `rtl_store_get_storesys_id_from_desc()` is successful, it returns the storage system ID.

If `rtl_store_get_storesys_id_from_desc()` fails, it returns -1.
**rtl_store_get_storesys_user_data_from_desc()**

This function returns the user data for the associated storage system.

**Syntax**

```c
void *rtl_store_get_storesys_user_data_from_desc(
    void *store);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The descriptor of the store.

**Return values**

If `rtl_store_get_storesys_user_data_from_desc()` is successful, it returns the storage system user data.

If `rtl_store_get_storesys_user_data_from_desc()` fails or no user data has been allocated, it returns null.
**rtl_store_get_symbol_set_size_from_desc()**

This function returns the symbol set size from the store descriptor.

The symbol set size is the value of the STOREXX_SYMBOL_SET_SIZE parameter in the configuration file.

**Syntax**

```c
int rtl_store_get_symbol_set_size_from_desc(void *store)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The store descriptor.

**Return values**

If the `rtl_store_get_symbol_set_size_from_desc()` function is successful, it returns the symbol set size.

If the `rtl_store_get_symbol_set_size_from_desc()` function fails, it returns `RTL_FAILURE`. 
rtl_store_get_symbol_set_size_from_id()

This function returns the symbol set size from the store ID.

The symbol set size is the value of the STOREXX_SYMBOL_SET_SIZE parameter in the configuration file.

Syntax

int rtl_store_get_symbol_set_size_from_id(int store_id)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

store_id

The store ID.

Return values

If the rtl_store_get_symbol_set_size_from_id() function is successful, it returns the symbol set size.

If the rtl_store_get_symbol_set_size_from_id() function fails, it returns RTL_FAILURE.
**rtl_store_get_symbol_table_name_from_desc()**

This function returns the store symbol table name from the store descriptor.

The symbol table name is the STOREXX_SYMBOL_TABLE part of the store entry in the configuration file for the store.

**Syntax**

```c
char *rtl_store_get_symbol_table_name_from_desc(
    void *store_desc);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_desc*

The descriptor of the store.

**Return values**

If `rtl_store_get_symbol_table_name_from_desc()` is successful, it returns the store symbol table name.

If `rtl_store_get_symbol_table_name_from_desc()` fails, it returns a zero-length string.
rtl_store_get_symbol_table_name_from_id()

The rtl_store_get_symbol_table_name_from_id() function returns the store symbol table name from the store ID.

The symbol table name is the STOREXX_SYMBOL_TABLE part of the store entry in the configuration file for the store.

**Syntax**

```c
char *rtl_store_get_symbol_table_name_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`store_id`

The ID number of the store.

**Return values**

If rtl_store_get_symbol_table_name_from_id() is successful, it returns the store symbol name.

If rtl_store_get_symbol_table_name_from_id() fails, it returns a zero-length string.
**rtl_store_get_sysname_from_desc()**

This function returns the name of the storage system that is associated with the given store descriptor.

**Syntax**

```c
char *rtl_store_get_sysname_from_desc(
    void *store);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The descriptor of the store.

**Return values**

If `rtl_store_get_sysname_from_desc()` is successful, it returns the storage name.

If `rtl_store_get_sysname_from_desc()` fails, it returns a zero-length string.
**rtl_store_get_sysname_from_id()**

This function returns the name of the storage system that is associated with the given store ID.

**Syntax**

```c
char *rtl_store_get_sysname_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store_id

The ID number of the store.

**Return values**

If rtl_store_get_sysname_from_id() is successful, it returns the system storage name.

If rtl_store_get_sysname_from_id() fails, it returns a zero-length string.

If the store specified by the store ID is not configured, this function returns a zero-length string. You can use the rtl_store_is_configured() function to determine if a store for a particular store ID is configured.
**rtl_store_get_tick_set_size_from_desc()**

This function returns the tick set size from the store descriptor.

The tick set size is the value of the STOREXX_TICK_SET_SIZE parameter in the configuration file.

**Syntax**

```c
int rtl_store_get_tick_set_size_from_desc(void *store)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store  The store descriptor.

**Return values**

If the `rtl_store_get_tick_set_size_from_desc()` function is successful, it returns the tick set size.

If the `rtl_store_get_tick_set_size_from_desc()` function is fails, it returns RTL_FAILURE.
**rtl_store_get_tick_set_size_from_id()**

This function returns the tick set size from the store ID.

The tick set size is the value of the STOREXX_TICK_SET_SIZE parameter in the configuration file.

**Syntax**

```c
int rtl_store_get_tick_set_size_from_id(int store_id)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`store_id`

The store ID.

**Return values**

If the `rtl_store_get_tick_set_size_from_id()` function is successful, it returns the tick set size.

If the `rtl_store_get_tick_set_size_from_id()` function is fails, it returns RTL_FAILURE.
**rtl_store_get_tick_table_name_from_desc()**

This function returns the store tick table name from the store descriptor.

The tick table name is the STOREXX_TICK_TABLE part of the store entry in the configuration file of the store.

**Syntax**

```c
char *rtl_store_get_tick_table_name_from_desc( void *store_desc);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_desc*

The descriptor of the store.

**Return values**

On success, returns the store tick table name.

On failure, returns a zero-length string.
**rtl_store_get_tick_table_name_from_id()**

This function returns the store tick table name from the store ID.

The tick table name is the STOREXX_TICK_TABLE part of the store entry in the configuration file of the store.

**Syntax**

```c
char *rtl_store_get_tick_table_name_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store_id

The ID number of the store.

**Return values**

If `rtl_store_get_tick_table_name_from_id()` is successful, it returns the store tick table name.

If `rtl_store_get_tick_table_name_from_id()` fails, it returns a zero-length string.
**rtl_store_get_user_data_from_desc()**

This function returns the store user data from the store descriptor.

This user data can be setup by the rtl_store_open() function implementation.

**Syntax**

```c
void *rtl_store_get_user_data_from_desc(
    void *store);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- `store` The descriptor of the store.

**Return values**

If rtl_store_get_user_data_from_desc() is successful, it returns the user data.

If rtl_store_get_user_data_from_desc() fails or no user data has been allocated, it returns null.
**rtl_store_is_configured()**

This function enables you to determine if a particular store is defined in the configuration file.

You can use this function to check the validity of a store ID before calling other store accessor functions, such as the rtl_store_get_info() function.

**Syntax**

```c
int rtl_store_is_configured(
    int p_store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`p_store_id`

The ID number of the store.

**Return values**

The rtl_store_is_configured() function returns 1 if the store for the specified ID is configured. Otherwise, it returns 0.
rtl_storesys_get_conn_from_desc()

This function returns the storage system conn from the storage system descriptor.

This information is the STORESYSXX_CONN part of the storage system entry in the configuration file for the given storage system.

Syntax

```c
char *rtl_storesys_get_conn_from_desc(
    void *store_sys);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

store_sys

The descriptor of the storage system.

Return values

If rtl_storesys_get_conn_from_desc() is successful, it returns the storage system conn.

If rtl_storesys_get_conn_from_desc() fails, it returns a zero-length string.

Note: The information from the STORESYSXX_CONN part of the storage system entry is optional. If it is not provided, a zero-length string is returned.

Related concepts

"Storage system API" on page 290

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_get_conn_from_id()**

This function returns the storage system conn from the storage system ID.

**Syntax**

```c
char *rtl_storesys_get_conn_from_id(
    int store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store_id

The ID of the storage system.

**Return values**

If `rtl_storesys_get_conn_from_id()` is successful, it returns the storage system connection string.

If `rtl_storesys_get_conn_from_id()` fails, it returns a zero-length string.

If the storage system specified by the storesys ID is not configured, this function returns a zero-length string. You can use the `rtl_storesys_is_configured()` function to determine if a storage system for a particular storesys ID is configured.

**Note:** The information from the STORESYSXX_CONN part of the storage system entry is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

[“Storage system API” on page 290](#)

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
rtl_storesys_get_ha_from_id()

This function returns the value of the STORESYSXX_HA parameter in the configuration file. This parameter specifies whether a storage system is configured for high availability support.

Syntax

```c
int rtl_storesys_get_ha_from_id(
    int p_storesys_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

p_storesys_id

The ID number of the storage system.

Return values

If rtl_storesys_get_ha_from_id is successful, it returns:

1  If the storage system is configured for high availability support.
0  If the storage system is not configured for high availability support.

If the store specified by the storesys ID is not configured, this function returns -1. You can use the rtl_storesys_is_configured() function to determine if a storesys for a particular storesys ID is configured.

Related concepts

“Storage system API” on page 290
DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_get_id_from_desc()**

This function returns the storage system ID from the storage system descriptor.

This ID is the XX part of the storage system entry (for example, STORESYSXX_NAME) in the configuration file for the given storage system.

**Syntax**

```c
int rtl_storesys_get_id_from_desc(
    void* store_sys);
```

**Valid in client application?** | **Valid in storage system?** | **Valid in feed handler?** | **Valid in HA library?** | **Valid in transport?**
--- | --- | --- | --- | ---
No | Yes | No | No | No

**Arguments**

`store_sys`

The descriptor of the storage system.

**Return values**

If `rtl_storesys_get_id_from_desc()` is successful, it returns the storage system ID (0-99).

If `rtl_storesys_get_id_from_desc()` fails, it returns -1.

**Related concepts**

“Storage system API” on page 290

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
rtl_storesys_get_id_from_name()

This function returns the ID of the given storage system name.

It does this by comparing the name to the STORESYSXX_NAME variables found in the config file and then returns the XX as a number.

**Syntax**

```c
int rtl_storesys_get_id_from_name(
    char *storesys_name);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

- **storesys_name**
  
  The name of the storage system.

**Return values**

If `rtl_storesys_get_id_from_name()` is successful, it returns the storage system ID.

If `rtl_storesys_get_id_from_name()` fails, it returns -1.

**Related concepts**

- "Storage system API" on page 290
  
  DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_get_info_from_desc()**

This function returns the storage system info from the storage system descriptor.

This information is the STORESYSXX_INFO part of the storage system entry in the configuration file for the given storage system.

**Syntax**

```c
char *rtl_storesys_get_info_from_desc(
    void *store_sys);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_sys*

The descriptor of the storage system.

**Return values**

If `rtl_storesys_get_info_from_desc()` is successful, it returns the storage system information.

If `rtl_storesys_get_info_from_desc()` fails, it returns a zero-length string.

**Note:** The information from the STORESYSXX_INFO part of the storage system entry is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

- [“Storage system API” on page 290](#)

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_get_info_from_id()**

This function returns the storage system info from the storage system ID.

This information is the STORESYSXX_INFO part of the storage system entry in the configuration file for the given storage system.

**Syntax**

```c
char *rtl_storesys_get_info_from_id(
    int `store_id`);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`store_id`

The ID of the storage system.

**Return values**

If `rtl_storesys_get_info_from_id()` is successful, it returns the storage system information.

If `rtl_storesys_get_info_from_id()` fails, it returns a zero-length string.

If the storage system specified by the storesys ID is not configured, this function returns a zero-length string. You can use the `rtl_storesys_is_configured()` function to determine if a storage system for a particular storesys ID is configured.

**Note:** The information from the STORESYSXX_INFO part of the storage system entry is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

- "Storage system API" on page 290

  DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
rtl_storesys_get_name_from_desc()

This function returns the storage system name from the storage system descriptor.

This name is the STORESYSXX_NAME part of the storage system entry in the configuration file for the given storage system.

**Syntax**

```c
char *rtl_storesys_get_name_from_desc(
    void *store_sys);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

store_sys

The descriptor of the storage system.

**Return values**

If rtl_storesys_get_name_from_desc() is successful, it returns the storage system name.

If rtl_storesys_get_name_from_desc() fails, it returns a zero-length string.

**Related concepts**

“Storage system API” on page 290

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_get_name_from_id()**

This function returns the name of the storage system with the given ID.

The ID refers to the storage system number used in the config file.

**Syntax**

```c
char *rtl_storesys_get_name_from_id(
    int    store_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*store_id*

The ID of the storage system.

**Return values**

If `rtl_storesys_get_name_from_id()` is successful, it returns the storage system name.

If `rtl_storesys_get_name_from_id()` fails, it returns a zero-length string.

If the storage system specified by the storesys ID is not configured, this function returns a zero-length string. You can use the `rtl_storesys_is_configured()` function to determine if a storage system for a particular storesys ID is configured.

**Related concepts**

[“Storage system API” on page 290](#)

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_get_password_from_desc()**

This function returns the unencrypted storage system password from the storage system descriptor.

This information is the STORESYSXX_PASSWORD part of the storage system entry in the configuration file for the given storage system.

**Syntax**

```c
char *rtl_storesys_get_password_from_desc(
    void *store_sys,
    char *pass_buf);
```

**Valid in client application?** | **Valid in storage system?** | **Valid in feed handler?** | **Valid in HA library?** | **Valid in transport?**
--- | --- | --- | --- | ---
No | Yes | No | No | No

**Arguments**

- **store_sys**
  - The descriptor of the storage system.

- **pass_buf**
  - A string buffer that is supplied by the caller. This buffer must be at least 17 bytes long (to hold a 16-character password and a null terminator).

**Return values**

If `rtl_storesys_get_password_from_desc()` is successful, it returns the buffer that was passed to it, with the unencrypted password in it.

If `rtl_storesys_get_password_from_desc()` fails, this function returns a zero-length string.

**Note:** The information from the STORESYSXX_PASSWORD part of the storage system entry is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

"Storage system API" on page 290

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_get_password_from_id()**

This function returns the unencrypted storage system password from the storage system ID.

This information is the STORESYSXX_PASSWORD part of the storage system entry in the configuration file for the given storage system.

**Syntax**
```c
char *rtl_storesys_get_password_from_id(
    int `store_id`
    char *pass_buf);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

**store_id**

The ID of the storage system.

**pass_buf**

A string buffer that is supplied by the caller. This buffer must be at least 17 bytes long (to hold a 16-character password and a null terminator).

**Return values**

If `rtl_storesys_get_password_from_id()` is successful, it returns the buffer that was passed to it, with the unencrypted password in it.

If `rtl_storesys_get_password_from_id()` fails, this function returns a zero-length string.

If the storage system specified by the storesys ID is not configured, this function returns a zero-length string. You can use the `rtl_storesys_is_configured()` function to determine if a storage system for a particular storesys ID is configured.

**Note:** The information from the STORESYSXX_PASSWORD part of the storage system entry is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

"Storage system API" on page 290

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
rtl_storesys_get_user_from_desc()

This function returns the storage system user from the storage system descriptor.

This information is the STORESYSXX_USER part of the storage system entry in the configuration file for the given storage system.

Syntax

```c
char *rtl_storesys_get_user_from_desc(
    void *store_sys);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

store_sys

The descriptor of the storage system.

Return values

If rtl_storesys_get_user_from_desc() is successful, it returns the storage system user.

If rtl_storesys_get_user_from_desc() fails, it returns a zero-length string.

Note: The information from the STORESYSXX_USER part of the storage system entry is optional. If it is not provided, a zero-length string is returned.

Related concepts

“Storage system API” on page 290

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_get_user_from_id()**

This function returns the storage system user from the storage system ID.

This information is the STORESYSXX_USER part of the storage system entry in the configuration file for the given storage system.

**Syntax**

```c
char *rtl_storesys_get_user_from_id(
    int `store_id`);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

`store_id`

The ID of the storage system.

**Return values**

If `rtl_storesys_get_user_from_id()` is successful, it returns the storage system user name.

If `rtl_storesys_get_user_from_id()` fails, it returns a zero-length string.

If the storage system specified by the storesys ID is not configured, this function returns a zero-length string. You can use the `rtl_storesys_is_configured()` function to determine if a storage system for a particular storesys ID is configured.

**Note:** The information from the STORESYSXX_USER part of the storage system entry is optional. If it is not provided, a zero-length string is returned.

**Related concepts**

- "Storage system API" on page 290

  DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_is_configured()**

This function enables you to determine if a particular storage system is defined in the configuration file. You can use this function to check the validity of a storage system ID before calling other storage system accessor functions, such as the rtl_storesys_get_info() function.

**Syntax**

```c
int rtl_storesys_is_configured(
    int p_storesys_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*p_storesys_id*

The ID number of the storage system.

**Return values**

The rtl_storesys_is_configured() function returns 1 if the storage system for the specified ID is configured. Otherwise, it returns 0.

**Related concepts**

"Storage system API" on page 290

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_storesys_log()**

This function provides a means for messages to be displayed in DSE message log.

The message file that a storage system uses is defined by the variable STORESYSXX_MSG_PATH in the config file.

**Syntax**

```c
void rtl_storesys_log(
    int storesys_id,
    int p_errno,
    ...
);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Arguments**

*storesys_id*

The ID of the storage system.

*p_errno*

The error number from the message file that should be printed.

*...*

A printf-style variable-length list of arguments corresponding to parameters in the log message.

**Return values**

The rtl_storesys_log() function has no return values.

**Related concepts**

"Storage system API" on page 290

DSE provides this API for accessing information about a storage system from either the storage system descriptor or a storage system ID.
**rtl_timet_to_rtltime()**

This function converts a timestamp from a time_t type to an rtl_datetime type.

The time_t type is a C date time representation with year to second precision defined in time.h. The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
void rtl_timet_to_rtltime(
    time_t t,
    rtl_datetime *rtl_time);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **t**: The timestamp as a time_t type (input parameter).
- **rtl_time**: The timestamp as an rtl_datetime type (output parameter).

**Return values**

The rtl_timet_to_rtltime() function has no return values.

**Related concepts**

- "Time API" on page 291
- DSE provides API functions for managing timestamps.
**rtl_tm_to_rtltime()**

This function converts a timestamp from a tm struct type to an rtl_datetime type.

The tm struct is a C date time representation with year to second precision defined in time.h. The rtl_datetime type is a DSE 64-bit date time representation with year to 10 microseconds precision.

**Syntax**

```c
void rtl_tm_to_rtltime(
    struct tm tm,
    rtl_datetime *rtl_time);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

- **tm** The timestamp as a tm struct type (input parameter).
- **rtl_time** The timestamp as an rtl_datetime type (output parameter).

**Return values**

The rtl_tm_to_rtltime() function has no return values.

**Related concepts**

- "Time API" on page 291
  - DSE provides API functions for managing timestamps.
**rtl_transport_get_info()**

This function returns the value of the TRANSPORTXX_INFO configuration parameter.

**Syntax**

```c
char * rtl_transport_get_info(int transport_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

*transport_id (in)*

The ID number of the transport.

**Return values**

If `rtl_transport_get_info()` is successful, it returns the TRANSPORTXX_INFO string.

If no TRANSPORTXX_INFO parameter is defined for this transport, or if no transport is configured for the specified ID, this function returns a zero-length string. Use the `rtl_transport_is_configured()` function to determine if a transport is configured for a particular transport ID.

**Related concepts**

[“Transport API” on page 292](#)

DSE provides this API for transport handlers to retrieve information about the transport.
rtl_transport_get_name()

This function returns the value of the TRANSPORTXX_NAME configuration parameter, where XX is the ID of the specified transport.

**Syntax**

```c
int rtl_transport_get_name(int transport_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

*transport_id (in)*

The ID number of the transport.

**Return values**

If rtl_transport_get_name() is successful, it returns the TRANSPORTXX_NAME string.

If rtl_transport_get_name() fails, or if no transport is configured for the specified ID, it returns a zero-length string. Use the rtl_transport_is_configured() function to determine if a transport for a particular transport ID is configured.

**Related concepts**

- [Transport API](#) on page 292
- DSE provides this API for transport handlers to retrieve information about the transport.
**rtl_transport_get_user_data()**

This function returns a transport handler's user data. The transport user data is a memory buffer allocated by the transport, and is returned to DSE by rtl_transport_handler_init(). The contents and use of this buffer are determined by the transport implementation.

**Syntax**

```c
void * rtl_transport_get_user_data(int transport_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

transport_id (in)

The ID number of the transport.

**Return values**

If rtl_transport_get_user_data() is successful, it returns a pointer to the transport handler user data area.

If rtl_transport_get_user_data() fails, or if no user data area has been allocated, it returns NULL.

**Related concepts**

- “Transport API” on page 292

DSE provides this API for transport handlers to retrieve information about the transport.
**rtl_transport_is_configured()**

This function enables you to determine if a particular transport is defined in the configuration file. Use this function to check the validity of a transport ID before calling other transport accessor functions such as rtl_transport_get_info().

**Syntax**

```c
int rtl_transport_is_configured(int transport_id);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

`transport_id` (in)

The ID number of the transport.

**Return values**

- 1 If the transport is configured for the specified ID.
- 0 If no transport is configured for the specified ID.

**Related concepts**

[“Transport API” on page 292](#)

DSE provides this API for transport handlers to retrieve information about the transport.
**rtl_transport_log()**

This function allows functions to write messages to the log file.

The message catalog that a transport uses to format log entries is defined by the value of the TRANSPORTXX_MSG_PATH parameter in the configuration file.

**Syntax**

```c
void rtl_transport_log(
    int transport_id,
    int msg_num,
    ...);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Arguments**

transport_id (in)

The ID number of the transport.

msg_num (in)

The number of the message from the transport message catalog that should be printed.

... (in)  A printf-style variable-length list of arguments corresponding to parameters in the log message.

**Related concepts**

“Transport API” on page 292

DSE provides this API for transport handlers to retrieve information about the transport.
rtl_unpack_nullbits()

This function converts a bitwise array to a byte array.

Each bit in the bitwise array is converted to a byte in the byte array.
- Each bit of value 0 is converted to a byte with value RTL_FIELD_IS_NOT_NULL.
- Each bit of value 1 is converted to a byte with value RTL_FIELD_IS_NULL.

This function can be called by the storage system implementation to convert a bitwise array of null indicators to byte array of null indicators.

Syntax

```c
void rtl_unpack_nullbits(
    char *bytearray,
    unsigned char *bitarray,
    int ncol);
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Arguments

bytearray (in/out)

A pointer to the byte array to hold the converted output.

bitarray (in)

A pointer to the bitwise array.

ncol (in)

The number of bits in the bitwise array.

Related concepts

"Record type API" on page 289

DSE provides API functions for retrieving information about record types and setting default values for record types.
Chapter 28. Public header files

DSE provides header files for developers of DSE components and applications.

The following table lists the public header files and their associated APIs.

<table>
<thead>
<tr>
<th>Public header file</th>
<th>Associated API</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl_client.h</td>
<td>C client</td>
</tr>
<tr>
<td>rtl_gen_halib.h</td>
<td>High availability</td>
</tr>
<tr>
<td>rtl_gen_store.h</td>
<td>Storage system</td>
</tr>
<tr>
<td>rtl_handler.h</td>
<td>Feed handler</td>
</tr>
<tr>
<td>rtl_rectype.h</td>
<td>All</td>
</tr>
<tr>
<td>rtl_state.h</td>
<td>High availability</td>
</tr>
<tr>
<td>rtl_time.h</td>
<td>All</td>
</tr>
<tr>
<td>rtl_trace.h</td>
<td>Feed handler, high availability, storage system, transport</td>
</tr>
<tr>
<td>rtl_transport.h</td>
<td>Feed handler, transport</td>
</tr>
</tbody>
</table>

*rtl_client.h*

Defines constants, types, and functions for use by C client applications.

*rtl_gen_halib.h*

Defines constants and functions for use by high availability communication handlers.

*rtl_gen_store.h*

Defines constants, types, and functions for use by generic storage systems.

*rtl_handler.h*

Defines constants, types, and functions for use by feed handlers.

*rtl_rectype.h*

Defines constants, types and functions to describe and work with tick and metadata record types, for use by C client applications and all server subsystems (feed handlers, storage systems, and high availability communication handlers).

*rtl_state.h*

Defines constants and functions to discover and change the state of DSE, for use by all server subsystems.

*rtl_time.h*

Defines types and functions for manipulating timestamps, for use by C client applications and all server subsystems.

*rtl_trace.h*

Defines constants and macros for adding tracing, for use by all server subsystems.

*rtl_transport.h*

Defines constants, types, data structures, and functions for use by transports.
Related concepts

Chapter 23, “Feed handlers,” on page 153
DSE uses feed handlers to receive and process data from different kinds of data feeds. The feed handler defines how the data is formatted and processed for the data feed.

Chapter 24, “Transports,” on page 163
The transport handles the feed communication transport. It simplifies the development effort required to create a feed handler by encapsulating the details of the feed transport mechanism.

Related tasks

Chapter 26, “Writing client applications,” on page 275
You can write C or Java client applications that connect directly to DSE to query and update data and retrieve statistics.
Part 6. Troubleshooting

DSE provides resources to help you troubleshoot problems, including a diagnostic log file, a trace mechanism, and a utility to examine DSE shared memory.
Chapter 29. DSE diagnostic log file

All messages are written to the DSE diagnostic log file to help you monitor your DSE configuration and diagnose problems.

DSE generates messages that consist of three elements separated by a space. These elements are:

- The time stamp, enclosed in square brackets. The time stamp is in ISO 8601 format and can be configured to use UTC or the local time zone. See the RTL_LOG_LOCALTIME configuration parameter in “General system configuration parameters” on page 52.
- The message ID.
- The message text. This text can contain spaces.

The message ID has the following format:

\[DSECXXXXY\]

where:

**DSE** Indicates that the error is from the DSE server.
**C** Indicates the component that generated the message.
- **F** Feed handler library
- **H** High availability communication library
- **I** Internal (core DSE)
- **S** Storage system library
- **T** Transport

**XXXX** Indicates the message number.

**Y** Indicates the severity of the message.
- **E** Error
- **F** Fatal
- **I** Information
- **W** Warning

A fatal error causes DSE to exit immediately without flushing data in memory to disk.

**Examples:**

The following example is an informational message that is related to the feed handler. This message uses a UTC time stamp.

\[2005-01-31T18:18:33Z\] DSEF0807I: Reader thread 278546 connected to source 239.255.0.3:ISE.

The following example is an informational message from the core DSE server. This message uses a timestamp in the local time zone.

\[2005-11-02T16:38:30-07:00\] DSEI0115I: Initialized 0 Symbols for Store 1
All messages are written to a single log file. The log file name and location are specified by the RTL_LOG_PATH parameter in the configuration file.
Chapter 30. Messages

Message information describes the cause of a message and any action you should take in response to the message.

The DSE messages are organized by type.

Server messages

These are the internal (core DSE) messages. The messages are listed in numeric sequence.

DSEI0000I Data Stream Engine

Explanation:
This message is logged when DSE starts. It displays the product version number and compilation date.

DSEI0001I DSE Execution completes.

Explanation:
This message is logged after DSE shuts down.

DSEI0002I Debug Message: text.

Explanation:
This message is for internal debugging purposes.

DSEI0003I Initialized feed feed_number with stamps

Explanation:
The rolling window of time was initialized to store ticks from this feed that are between timestamp1 and timestamp2. The size of the window is determined by the configuration and rolls forward as time progresses.

DSEI0004F Failed to install signal handler for

Explanation:
DSE cannot install the signal handler for the specified signal.

User response:
Contact IBM Software Support.

DSEI0005I Re-reading config file filename.

Explanation:
DSE is re-reading the configuration file filename because the following command was issued: rtlmode -r

DSEI0006W Failed to re-read config file.

Explanation:
The configuration file cannot be read.

User response:
See the log file for the preceding message or messages that indicate the source of the error. These messages appear after the DSEI0005 log entry.

The log file is in the directory that is specified by the RTL_LOG_PATH parameter in the configuration file.

DSEI0007F A major failure occurred; shutting down immediately.

User response:
See the log file for the preceding message or messages that indicate the source of the error.

DSEI0008I Cancel thread_name thread thread_id.

Explanation:
This message is logged when DSE terminates each thread.

DSEI0009E Cannot remove message queue

Explanation:
The DSE server cannot remove a message queue with the ID message_queue_identifier using the queue key key_value The key_value key is the same as the shared memory for this DSE server instance. The rtlmode and rtlstat utilities use this message queue to communicate with the DSE server. errno indicates the error number that was returned by the msgct1 operating system call. errno_text is a text string describing errno.

User response:
If necessary, manually remove the message queue by running the ipcrm command.

DSEI0010F Wrote number_symbols symbols, and

Explanation:
number_ticks ticks to shared memory.
DSEI0011E • DSEI0018F

**Explanation:**
During shutdown, DSE logs this message to record the total number of symbols and ticks that were inserted into shared memory after DSE started.

DSEI0011E Message message_number cannot be found in subsystem message file.

**Explanation:**
A subsystem (a feed, feed handler, storage system, or the high availability communications library) called one of the logging API functions with an unknown message number.

**User response:**
If the message number is outside of the range of supported values (0 to 999, inclusive), recode the subsystem to use message numbers that are in the supported range. If the message number is within the supported range, ensure that the message file is correct and that it contains all of the necessary messages.

DSEI0012I Caught signal signal_number: (action).

**Explanation:**
An external signal was sent to DSE. action can be:

- handled
  - DSE takes action in response to the signal.

- ignored
  - DSE ignores the signal and does not take any action in response to it.

DSEI0013I Copy buckets from feed feed_id1 to feed_id2.

**Explanation:**
The rolling window of memory for ticks from feed_id2 was initialized to the same timestamps as feed_id1.

DSEI0014E Messages between server and client are out of sequence.

**Explanation:**
An application using the DSE client API for C or Java sent an unrecognizable request to the DSE server.

**User response:**
Ensure that the version of the DSE client API matches the version of the DSE server. Ensure that your application calls the DSE client API functions in the correct sequence.

DSEI0015F Error opening Shared Memory. errno=errno (errno_text).

**Explanation:**
DSE cannot allocate a shared memory segment. errno indicates the error number that was returned by the shmat operating system call. errno_text is a text string describing errno.

**User response:**
Increase the amount of available shared memory on the system.

If errno is 22 (EINVAL), determine if an existing shared memory segment is using the same shared memory key (specified by the SHMKEY parameter) with a different size (specified by the SHMSIZE parameter), by running the following command: 

```bash
ipcs -m
```

If this additional shared memory segment exists, remove it by running the `ipcrm -m id` command. Then restart DSE.

DSEI0016F Error attaching to Shared Memory.

**Explanation:**
DSE cannot attach to a shared memory segment. errno indicates the error number that was returned by the shmat operating system call. errno_text is a text string describing errno.

**User response:**
Check the shared memory key that is being used and restart DSE. If the error persists, contact IBM Software Support.


**Explanation:**
The size of shared memory segments is limited to maximum_bytes (generally 1 gigabyte on most platforms). The first memory segment holds a number of data structures that DSE uses internally. The total size of these data structures must not exceed maximum_bytes. The size of two of these data structures can be controlled by the following configuration parameters:

- HASHSIZE
  - Determines the size of the symbol hash table.

- TRACEBUFFER_SIZE
  - Determines the size of the tracing buffers.

**User response:**
Reduce the value of the HASHSIZE or TRACEBUFFER_SIZE parameter, or both, so that the size of the first memory segment is less than maximum_bytes.

DSEI0018F Tick pool too small, min. shmem size [needed_bytes] current [actual_bytes].

**Explanation:**
Not enough shared memory is available to accommodate the tick pool. The tick pool holds all incoming ticks.

**User response:**
Increase the value of the SHMSIZE parameter so that it is greater than or equal to \textit{needed\_bytes}.

DSEI0019E  Security Header pool is full.

Explanation:
Not enough shared memory is available in the security header pool to accommodate new symbols. The security header pool holds information about each symbol (such as security information).

User response:
Ensure that the value of the SECHDR\_SIZE parameter is large enough to accommodate all existing and anticipated new symbols in all stores.

DSEI0020W  \textit{function\_name} Missed \textit{[number\_locks]} locks for \textit{lock\_type}.

Explanation:
Two or more DSE threads have an unusually high amount of lock contention between them.

User response:
Depending on which threads are competing for the same lock, try these actions:
\begin{itemize}
  \item Increase the number of flushers.
  \item Reconfigure the storage system so that flushes can be completed more quickly.
  \item Change the purging parameters to purge less often.
\end{itemize}

DSEI0021I  Dynamically adding \textit{number\_bytes} bytes of shared memory.

Explanation:
When the percentage utilization of memory that contains ticks exceeds the value of the PURGE\_HWM parameter, DSE adds shared memory segments to accommodate the incoming ticks. This message is logged as each new shared memory segment is allocated.

DSEI0022E  Store \textit{(store\_name)} Record Type \textit{(record\_id)} Out of Range (0-99).

Explanation:
The value of the STOREXX\_RECTYPE parameter for the specified store is outside of the range of supported values.

User response:
Ensure that the STOREXX\_RECTYPE parameter is set to the ID of an existing RECORDXX\_SCHEMA or RECORDXX\_META\_SCHEMA parameter definition.

DSEI0023E  Store \textit{(store\_name)} Record Type \textit{(record\_id)} not found.

Explanation:
The value of the STOREXX\_RECTYPE parameter for the specified store refers to an undefined record type.

User response:
Ensure that the STOREXX\_RECTYPE parameter is set to the ID of an existing RECORDXX\_SCHEMA or RECORDXX\_META\_SCHEMA parameter definition.

DSEI0024E  Cannot open message file \textit{message\_file\_name}.

Explanation:
The specified message file cannot be found, or cannot be opened for read access.

User response:
Ensure that the message file exists and that DSE can read it. Ensure that the file name is correct. If the file name is a relative path, ensure that the path is relative to the location of the configuration file.

DSEI0025E  Error at line \textit{line\_number} in message file \textit{file\_name}. Reason=\textit{reason\_code}.

Explanation:
The message at the specified line number in the message file is not formatted correctly.

The reason codes are:
\begin{itemize}
  \item 1  The message is longer than 512 bytes.
  \item 2  The message contains an unsupported severity code.
  \item 3  The message number is outside of the range of supported values (0 to 999, inclusive).
\end{itemize}

User response:
Ensure that all messages have the format \textit{number.severity.body}, where
\begin{itemize}
  \item \textit{number} is an integer between 0 and 999, inclusive.
  \item \textit{severity} is I, W, E or F.
  \item \textit{body} is the body of the message, which must be less than 500 bytes long.
\end{itemize}

DSEI0026F  Could not find free tail offset \textit{[freelist\_tail]}.

Explanation:
An internal error occurred. DSE cannot access the list of free ticks in shared memory.

User response:
Restart DSE. If the error persists, contact IBM Software Support.

DSEI0027W  Free list head and tail mis-aligned, \textit{[freelist\_head]} \textit{[freelist\_tail]}.

Explanation:
The list of free ticks that is in shared memory is in an inconsistent state.

User response:
DSEI0028F • DSEI0036E

Restart DSE. If the error persists, contact IBM Software Support.

DSEI0028F  Memory allocation failed (number_bytes bytes at line line_number in file source_code_file).

Explanation:
DSE has insufficient virtual memory. line_number and source_code_file indicate the location in DSE code where the error occurred.

User response:
Increase the amount of private virtual memory that is available to a process. Run the UNIX ulimit command to increase the maximum size of the data area for a user process.

DSEI0029E  Error getting nulls: record type record_id invalid.

Explanation:
The null indicator information for a tick or symbol cannot be accessed.

User response:
Ensure that RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definitions exist for the specified record_id.

DSEI0030E  PURGE_LWM [lwm_value] must be less than PURGE_HWM [hwm_value].

Explanation:
The value of the PURGE_LWM parameter must be less than the value of the PURGE_HWM parameter.

User response:
Ensure that the values of the PURGE_LWM and PURGE_HWM parameters are both between 1 and 99 inclusive. Ensure that the value of the PURGE_LWM parameter is less than the value of the PURGE_HWM parameter.

DSEI0031I  Purge watermark_type changed: [old_value]=>[new_value].

Explanation:
The value of the PURGE_LWM or PURGE_HWM parameter was changed as a result of DSE rereading the configuration file.

DSEI0032E  Error in feed feed_id: store Id store_id is out of range (0-99) or not defined.

Explanation:
The rtl_feed_get_tick_from_message function of the specified feed returned a tick to DSE with a store ID that is either:
• Outside of the range of supported values (0 to 99, inclusive).

• Within the range of supported values, but the store number is not defined in the DSE configuration file.

User response:
Ensure the feed handler returns the correct tick and metadata store IDs to DSE.

DSEI0033E  Error getting nulls for flush tick (store_name).

Explanation:
The null indicator information for a tick cannot be accessed.

User response:
Use this message in conjunction with the DSEI0029E message to identify the incorrect store and record type configuration. Ensure that the STOREXX_RECTYPE parameter for the specified store corresponds to a RECORDXX_SCHEMA parameter definition.

DSEI0034E  Error setting nulls: record type record_id invalid.

Explanation:
The null indicator information for a tick or symbol cannot be set.

User response:
Ensure that RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition exists for the specified record_id.

DSEI0035E  Error setting nulls: field field_number should never be null (record type record_id).

Explanation:
An attempt was made to set the value of a tick or metadata field to null, in violation of a NOT NULL constraint for that field.

User response:
Ensure that the feed handler is aware of which tick and metadata fields are allowed to be null. Ensure that the NOT NULL constraints in the specified RECORDXX_SCHEMA and RECORDXX_META_SCHEMA parameter definitions are correct.

DSEI0036E  Could not open config file filename for reading.

Explanation:
The config file specified by an INCLUDE parameter cannot be opened.

User response:
Verify that the file exists and is readable. If the filename is a relative path, verify that the path is relative to the location of the main config file.
DSEI0037I  No Disk write changed: 
[old_value] => [new_value].

Explanation:
The NODISKWRITE parameter was changed because DSE reread the configuration file.

DSEI0038I  New symbol enable flag changed: 
[old_value] => [new_value].

Explanation:
The ENABLE_NEWSYM parameter was changed because DSE reread the configuration file.

DSEI0039E  Cannot open log file file_name.
Explanation:
The specified log file cannot be opened for write access.
User response:
Ensure that the directory containing the log file has sufficient permissions for DSE to create a log file. If the log file already exists, ensure that DSE can write to it. If the filename is a relative path, ensure that the path is relative to the location of the configuration file.

DSEI0040I  Flusher thread thread_id begins shutdown.

Explanation:
The specified flusher thread flushed all of the remaining ticks to the store and is starting to shut down. DSE logs this message when it shuts down after the rtlmode -k command is run.

DSEI0041I  thread_name thread thread_id exited.
Explanation:
This message is logged by each thread as DSE shuts down.

DSEI0042F  Error Opening Handler Library 
(handler_name) (library_path) : error_info.

Explanation:
The dynamic library for the specified feed handler cannot be opened. error_info provides information on the type of error that was encountered.
User response:
Ensure that the HANDLERXX_LIB_PATH parameter for the handler library is correct. If the value of this parameter is a relative path, the library must be located in a directory that is included in the list of directories that the LD_LIBRARY_PATH environment variable specifies.

DSEI0044F  Error Initializing Handler (handler_name): 
error_code.

Explanation:
The rtl_handler_init function of the feed handler returned an error to the DSE server.
User response:
See the log file for a preceding message from the feed handler for more information.

DSEI0045E  Error Shutting Down Handler 
(handler_name): error_code.

Explanation:
The rtl_handler_shutdown function of the feed handler returned an error to the DSE server.
User response:
See the log file for a preceding message from the feed handler for more information.

DSEI0046E  Error Closing Handler Library 
(handler_name) (library_path) : error_info.

Explanation:
The dynamic library for the specified feed handler cannot be closed. error_info provides information on the type of error that was encountered.
User response:
Contact IBM Software Support.

DSEI0047F  Error Feed feed_id Handler Name Not Found (handler_name).

Explanation:
The feed handler for the specified feed cannot be found.
User response:
Ensure that FEEDXX_HANDLER_NAME is defined for the feed. Ensure that the name matches the value of the corresponding HANDLERXX_NAME parameter.

DSEI0048E  Error retrieving or creating Metadata for Symbol symbol_name (store_id:feed_id).

Explanation:
The metadata area for a symbol cannot be found or created. This metadata is for the given symbol_name symbol in the store_id store that is received by the feed_id feed.
User response:
Check for related errors in the log file. The security header pool might be full; if it is, increase the value of the SECHDR_SIZE configuration parameter. If you cannot see any related errors, contact IBM Software Support.
DSEI0049E  Error Initializing Feed (feed_id): error_code.

Explanation: The rtl_feed_init function of the feed handler returned an error to the DSE server.

User response: See the log file for a preceding message from the feed for more information.

DSEI0050E  Error Shutting Down Feed (feed_id): error_code.

Explanation: The rtl_feed_shutdown function of the feed handler returned an error to the DSE server.

User response: See the log file for a preceding message from the feed for more information.

DSEI0051E  Error opening source for feed feed_id: error_code.

Explanation: The rtl_transport_source_open function of the transport returned an error to the DSE server.

User response: See the log file for a preceding message from the transport for more information.

DSEI0052I  Loaded number_symbols symbols into memory.

Explanation: When DSE starts, it loads an in-memory symbol table of all of the existing symbols by reading the corresponding symbol table for each store. This message is logged after all of the symbols for all of the stores are loaded.

DSEI0053E  Error closing source for feed feed_id: error_code.

Explanation: The rtl_transport_source_close function of the transport returned an error to the DSE server.

User response: See the log file for a preceding message from the transport for more information.

DSEI0054E  Error getting message from source for feed feed_id: error_code.

Explanation: The rtl_feed_get_tick_from_message function of the feed handler returned an error to the DSE server.

User response: See the log file for a preceding message from the feed handler for more information.

DSEI0055E  Error updating metadata from tick for Feed (feed_id): error_code.

Explanation: The rtl_feed_update_metadata function of the feed handler returned an error to the DSE server.

User response: See the log file for a preceding message from the feed for more information.

DSEI0056E  At least one object_name must be configured.

Explanation: The configuration file defines too few parameters for DSE to start.

User response: Define at least one storage system, store, handler and feed in the configuration file, then restart DSE. See Chapter 9, “System configuration parameters,” on page 51 for more information about the parameters that are needed in the configuration file.

DSEI0057E  Cannot create message queue: key=key_value errno=errno (errno_text).

Explanation: The DSE server cannot create a message queue using key_value as the queue key. key_value is the same as the shared memory for this DSE server instance. This message queue is used by the rtlsmode and rtlsstat utilities to communicate with the DSE server. errno indicates the error number that was returned by the msgget operating system call. errno_text is a text string describing errno.

User response: Ensure that the shared memory key value is correct. If the system limit for the maximum number of message queues was exceeded, increase the limit.

DSEI0058E  Read error on message queue message_queue_identifier: key=key_value errno=errno (errno_text).

Explanation: The DSE server cannot read a message from the rtlsmode or rtlsstat utility. errno indicates the error number that was returned by the msgrcv operating system call. errno_text is a text string describing errno.

User response: Retry the rtlsstat or rtlsmode command that caused the error. If the error persists, restart DSE.
DSEI0059E  Write error on message queue
message_queue_identifier: key=key_value
errno=errno (errno_text).

Explanation:
The DSE server cannot send messages to the message queue using the queue key key_value. The key_value key is the same as the shared memory key for this DSE server instance. The rttmode and rttstat utilities use this message queue to communicate with the DSE server. errno indicates the error number that was returned by the msgsnd operating system call. errno_text is a text string describing errno.

User response:
Retry the rttstat or rttmode command that caused the error. If the error persists, restart DSE.

DSEI0060E  No shared memory left for tick symbol
[symbol_name], mem usage bytes_used,
tick size bytes_needed.

Explanation:
DSE lacks sufficient space in shared memory to add a tick for the specified symbol. The DSE server will try to add the tick again after a short delay (to give the purger an opportunity to free up space). After 100 unsuccessful retries, DSE shuts down.

User response:
Try these actions to accommodate the rate of incoming ticks:
- Increase the value of the MAX_SHMSIZE parameter to make more shared memory available to DSE.
- Decrease the value of the PURGE_HWM parameter to purge ticks more often.
- Increase the value of the STOREXX_NUM_FLUSHERS parameter to flush ticks to the store at a faster rate.
- Change the value of the REFRESH_INTERVAL or NUM_INTERVALS parameter, or both, to flush and purge ticks more often.

DSEI0061F  No more shared memory is available; shutting down.

Explanation:
DSE lacks sufficient space in shared memory to add a tick for the specified symbol. If memory is still not available after the purger attempts to free up space for additional ticks, DSE shuts down. This message is logged after the DSEI0060E message is logged.

User response:
Try these actions to accommodate the rate of incoming ticks:
- Increase the value of the MAX_SHMSIZE parameter to make more shared memory available to DSE.
- Decrease the value of the PURGE_HWM parameter to purge ticks more often.

DSEI0062W  Security header [symbol_name] not found.

Explanation:
The specified symbol is expected to be in DSE shared memory but cannot be found.

User response:
Contact IBM Software Support.

DSEI0063E  Unknown subsystem id [id_number]
passed to logging API function.

Explanation:
A subsystem (feed, feed handler, or storage system) called one of the logging API functions with an unknown feed ID, handler ID, or storage system ID.

User response:
Ensure that the feed ID, handler ID, or storage system ID corresponds to a defined feed, handler, or storage system, respectively.

DSEI0064W  Ignore symbol [symbol_name] from feed
[feed_id1] already had it down feed
[feed_id2].

Explanation:
The specified symbol being stored by feed feed_id1 is already stored by feed feed_id2. A symbol can be stored in a particular store by one feed only.

User response:
Ensure that messages for each symbol that is destined for a particular store are received on one feed only.

DSEI0065W  Obsolete config file parameter
parameter_name ignored.

Explanation:
The specified parameter is no longer needed.

User response:
Remove the parameter definition from the configuration file.

DSEI0066W  No value for parameter parameter_name
specified, using default [default_value].

Explanation:
This message is a warning that default values will be used.

User response:
Review this warning and ensure that the default values are appropriate for your configuration. If necessary, shut down DSE, specify a value, and restart DSE.
DSEI0067E  Duplicate config parameter [parameter_name].

Explanation:
The same parameter was defined more than once in the configuration file.

User response:
Eliminate duplicate parameter definitions from the configuration file and restart DSE.

DSEI0068E  Required parameter parameter_name is missing.

User response:
Define the missing parameter in the configuration file and restart DSE.

DSEI0069E  Transport transport_id: Value for property (property_name) must be between min_value and max_value inclusive.

Explanation:
The rtl_transport_get_property function of the transport returned an unexpected value. The value for the named property should be between the stated values.

User response:
See the log file for a preceding message from the transport for more information. Ensure that the transport is configured correctly. If you implemented the transport, then check that the rtl_transport_get_property function is implemented correctly.

DSEI0070E  number_symbols symbols could not be inserted in store (store_name).

Explanation:
In response to the rtlmode -k command to shut down, DSE attempts to insert all remaining new symbols to the store. If it is unable to do so after four attempts, it logs this message. The possible reasons for failed attempts are:

- The store is offline.
- The store rejected the insert attempt. (For example, if a store is a database, a constraint violation can cause this rejection).
- The connection to the store was lost.

User response:
See the log file for one or more preceding messages from the store to diagnose the cause of the insert failure.

DSEI0071I  Starting thread thread_id Flusher flusher_number of numberflushers for Store store_id (store_name).

Explanation:
This message is logged when each flusher thread starts up during DSE initialization.

DSEI0072E  Error registering source msg callback for feed feed_id: error_code.

Explanation:
The rtl_feed_source_register_callback function of the feed handler returned an error to the DSE server.

User response:
See the log file for a preceding message from the feed for more information.

DSEI0073E  Error getting tick from message for Feed (feed_id): error_code.

Explanation:
The rtl_feed_get_tick_from_message function of the feed handler returned an error to the DSE server.

User response:
See the log file for a preceding message from the feed for more information.

DSEI0074E  Failed to unlock thread_name mutex (source_code_file line line_number).

Explanation:
A thread mutex cannot be unlocked.

User response:
Contact IBM Software Support.

DSEI0075E  Failed to set up thread_name wait condition.

Explanation:
An internal error occurred. DSE cannot pause the named internal thread.

User response:
Restart DSE. If the error persists, contact IBM Software Support.

DSEI0076E  Failed to lock thread_name mutex (source_code_file line line_number).

Explanation:
A thread mutex cannot be locked.

User response:
Restart DSE. If the error persists, contact IBM Software Support.

DSEI0077E  Error getting message to publish for Feed (feed_id): error_code.

Explanation:
The rtl_feed_get_publish_message function of the feed handler returned an error to the DSE server.

User response:
See the log file for a preceding message from the feed handler for more information.

**DSE0078E**  Value for parameter parameter_name must be greater than or equal to min_value.

Explanation:  The specified numeric value is too small.

User response:  Increase the value of the parameter and restart DSE.

**DSE0079E**  Value for parameter parameter_name must be between min_value and maximum_value inclusive.

Explanation:  The numeric value for the specified parameter is outside of the range of supported values.

User response:  Change the value so that it is within the range of supported values and restart DSE.

**DSE0081I**  Command line: command_line_arguments.

Explanation:  This message is logged when DSE starts. It records the rconsole command line arguments for diagnostic purposes.

**DSE0082E**  Invalid hashid [hash_id] for symbol symbol_name.

Explanation:  An internal error occurred. DSE cannot correctly hash the specified symbol.

User response:  Restart DSE. If the error persists, contact IBM Software Support.

**DSE0083W**  Recommendation: flush period for store store_name should be less than maximum_interval seconds.

Explanation:  DSE uses a rolling window of time for storing ticks in shared memory. As the rolling window of time progresses, DSE frees the ticks that are no longer within the window so that new ticks from the feeds can be stored. Ticks cannot be freed from memory until they are flushed to the store. DSE flushes ticks from memory to the store periodically. The rate of flushing is determined by the store Flush Period parameter. The value of this parameter should be less than the value of the Refresh Interval parameter, so that DSE can keep the rolling window moving forward efficiently.

User response:

**DSE0084I**  number ticks initialized.

Explanation:  This message is logged when DSE initializes its shared memory at startup. It is logged once for every 256 K ticks that are initialized in the tick pool.

**DSE0085E**  Value for parameter parameter_name must be numeric.

Explanation:  The specified parameter requires a numeric value.

User response:  Ensure that the value contains only digits and that it is not enclosed in quotation marks.

**DSE0086W**  Value for parameter parameter_name is too small; adjusted from old_value to new_value.

Explanation:  DSE automatically increased the value of the parameter to a more appropriate value.

User response:  Review this warning and make sure the value that DSE chose is appropriate for your configuration. If necessary, shut down DSE, specify a new value, and restart DSE.

**DSE0087W**  Value for parameter parameter_name is too large; adjusted from old_value to new_value.

Explanation:  DSE automatically decreased the value of the parameter to a more appropriate value.

User response:  Review this warning and make sure the value that DSE chose is appropriate for your configuration. If necessary, shut down DSE, specify a new value, and restart DSE.

**DSE0088E**  Failed to add tick for symbol symbol_name. Errcode=error_code. (store_id:feed_id).

Explanation:  DSE cannot write a tick for the specified symbol to shared memory. error_code can be one of the following:

1  The symbol hash table is full.
2  The symbol header record (in the DSE security header pool) cannot be found. Or, if this is a new symbol, the symbol header record cannot be created.
The null indicators for the tick cannot be set.

-1 No more shared memory is available.

**User response:**
See the log file for a preceding message for additional information. Based on these messages and `error_code`, try these actions:

- If the symbol hash table is full, increase the value of the `HASHSIZE` parameter.
- If the null indicators cannot be set, ensure that the `STOREXX_RECTYPE` parameter for the specified store corresponds to a defined `RECORDXX_SCHEMA` definition. Ensure that the `NOT NULL` constraints in the schema definition are correct.
- If no more shared memory is available, increase the value of the `SHMSIZE`, `MAX_SHMSIZE`, or `ADD_SHMSIZE` parameter (or all of them) to accommodate additional ticks.

**Explanation:**
A `BYTE` or `CHAR` type must be qualified with a length specifier of the form "(n)" where n is a positive integer.

**User response:**
Correct the schema definition and restart DSE.

**Explanation:**
A `RECORDXX_SCHEMA` or `RECORDXX_META_SCHEMA` parameter definition must contain at least one field.

**User response:**
Correct the schema definition and restart DSE.

**Explanation:**
Each byte of the default value for a `BYTE` data type must be represented by two hexadecimal digits (0 to 9, A to F).

**User response:**
Correct the hexadecimal string representation of the incorrect field by adding or deleting characters as necessary.

**Explanation:**
The sequence definition contains mismatched parentheses or quotation marks.

**User response:**
Ensure that the sequence definition is enclosed in parentheses. If any field names contain special characters (such as punctuation or spaces), those characters must be properly escaped with backslashes.
**DSEI0097E**  Bad sequence (record type record_id):
Could not parse line: text.

**Explanation:**
The sequence definition does not contain any fields.

**User response:**
Ensure that the sequence definition consists of one or more field names that are separated by commas. These field names must correspond to fields in the associated RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition. The entire list of field names must be enclosed in parentheses.

**DSEI0098E**  Bad sequence (record type record_id):
Invalid field name 'field_name'.

**Explanation:**
The sequence definition has a field name that is not correctly specified.

**User response:**
Ensure that the sequence definition consists of one or more field names that are separated by commas. These field names must correspond to fields in the associated RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition. The entire list of field names must be enclosed in parentheses.

**DSEI0099E**  Bad sequence (record type record_id):
Field 'field_name' must have NOT NULL and WITH DEFAULT qualifiers.

**Explanation:**
All fields of a sequence definition must have both NOT NULL and WITH DEFAULT qualifiers in the corresponding RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition.

**User response:**
In the configuration file, correct the values of the RECORDXX_META_SCHEMA and RECORDXX_META_SEQUENCE parameter definitions to meet these constraints.

**DSEI0101E**  Bad sequence (record type record_id):
Unknown field name field_name.

**Explanation:**
A field name in a sequence definition does not match any field in the corresponding RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition.

**User response:**
Ensure that the field names of a sequence definition correspond to fields in the associated RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition.

**DSEI0102E**  Bad sequence (record type record_id):
Unexpected keyword or value found: text.

**Explanation:**
The sequence definition is incorrect.

**User response:**
Ensure that the sequence definition consists of one or more field names that are separated by commas. These field names must correspond to fields in the associated RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition. The entire list of field names must be enclosed in parentheses.

**DSEI0103I**  Thread thread_id has more ticks to flush.

**Explanation:**
DSE logs this message when it shuts down in response to the r1mode -k command. This message indicates that the specified flusher thread will start to flush all of the remaining ticks to the store before shutting down.

**DSEI0104E**  Bad schema (record type record_id, field field_number): Default Byte hexadecimal string could not be converted to binary.

**Explanation:**
A character other than 0 to 9 or A to F was encountered in the default value of a BYTE field.

**User response:**
Correct the hexadecimal string representation of the field so that it consists only of the digits 0 to 9 or the letters A to F. Lower case letters a to f are also allowed.

**DSEI0105F**  Feed feed_id transport name not found (transport_name).

**Explanation:**
The value of the FEEDXX_TRANSPORT_NAME parameter for the specified feed is not the name of a defined transport.

**User response:**
Ensure that the value of the FEEDXX_TRANSPORT parameter for the specified feed matches the name of
DSEI0106E • DSEI0114F

The associated transport (specified by the TRANSPORTXX_NAME parameter).

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Explanation</th>
<th>User response</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSEI0106E</td>
<td>Error beginning symbol add for store (store_name) error_code.</td>
<td>The rtl_store_symbol_add_begin function of the storage system returned an error to the DSE server.</td>
<td>See the log file for a preceding message from the storage system for more information.</td>
</tr>
<tr>
<td>DSEI0107E</td>
<td>Error ending symbol add for store (store_name) error_code.</td>
<td>The rtl_store_symbol_add_end function of the storage system returned an error to the DSE server.</td>
<td>See the log file for a preceding message from the storage system for more information.</td>
</tr>
<tr>
<td>DSEI0108E</td>
<td>Error beginning symbol add batch for store (store_name) error_code.</td>
<td>The rtl_store_symbol_add_batch_begin function of the storage system returned an error to the DSE server.</td>
<td>See the log file for a preceding message from the storage system for more information.</td>
</tr>
<tr>
<td>DSEI0109E</td>
<td>Error ending symbol add batch for store (store_name) error_code.</td>
<td>The rtl_store_symbol_add_batch_end function of the storage system returned an error to the DSE server.</td>
<td>See the log file for a preceding message from the storage system for more information.</td>
</tr>
<tr>
<td>DSEI0110E</td>
<td>Error Flushing to Store (store_name) (error_code) for flusher flusher_id.</td>
<td>The specified flusher cannot flush ticks to the store. error_code can be one of the following:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>An unrecoverable error occurred.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-5</td>
<td>An error occurred and DSE tried to flush ticks again. The STOREXX_RETRY_COUNT and STOREXX_RETRY_PERIOD parameters determine the number of attempts and the time interval between each attempt.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-6</td>
<td>The connection with the store was lost. DSE attempts to reconnect to the store. The</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-20</td>
<td>The Flush Count Error Symbol symbol_name in Store store_name (flush_count &gt; tick_count).</td>
<td>An internal error occurred. The flush state for the symbol is in an inconsistent state.</td>
</tr>
<tr>
<td>DSEI0113E</td>
<td>Error Flushing Ticks for Bucket bucket_number in Store store_name.</td>
<td>Ticks for the specified memory bucket cannot be flushed to the specified store. This error might occur while DSE is:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Writing tick data to the store.</td>
<td>• Committing a batch of ticks to the store.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Committing a batch of ticks to the store.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSEI0114F</td>
<td>Error Initializing SecHdr (symbol_name) symbol_id store_id).</td>
<td>The in-memory symbol table for the store cannot be initialized. Possible causes of this error are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Insufficient shared memory was allocated to store symbols.</td>
<td>• A problem occurred in the rtl_store_symbol_allocate_flusher routine. This routine is called if the STOREXX_ROUND_ROBIN parameter is set to 0.</td>
<td>Try these actions to fix the error:</td>
</tr>
</tbody>
</table>
• Ensure that the value of the SECHDR_SIZE parameter is large enough to accommodate all existing and anticipated new symbols in all stores.
• If the value of the STOREXX_ROUND_ROBIN parameter is 0, see the log file for a preceding message from the storage system.

DSEI0115I  Initialized number Symbols for Store store_id.

Explanation:
When DSE starts, it loads an in-memory symbol table of all of the existing symbols by reading the corresponding symbol table from the store. This message is logged once for every 100 000 symbols that are initialized.

DSEI0116F  Error Feed feed_id Store Name Not Found (store_name).

Explanation:
The value of FEEDXX_STORE_NAME for the specified feed is not the name of a defined store.

User response:
Ensure that the value of the FEEDXX_STORE_NAME parameter matches the value of the STOREXX_NAME parameter for the store that is associated with this feed. Note that the FEEDXX_STORE_NAME parameter is optional; a feed does not need to be associated with a particular store.

DSEI0117F  Error Store (store_name) Storage System Name Not Found (storage_system_name).

Explanation:
The storage system for the specified store cannot be found.

User response:
Ensure that the value of the STOREXX_SYSNAME for the specified store matches the name of its associated storage system (specified by the STORESYSXX_NAME parameter).

DSEI0118E  Error Opening Store (store_name) For Mode (mode_number) Id (flusher_number) error_code.

Explanation:
The rtl_store_open function of the storage system returned an error to the DSE server. flusher_number is meaningful only when the store is opened to flush ticks.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0119E  Error Closing Store (store_name) For Mode (mode_number) error_code.

Explanation:
The rtl_store_close function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0120E  Error Beginning Symbol Scan For Store (store_name) error_code.

Explanation:
The rtl_store_symbol_begin_scan function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0121E  Error Getting Next Symbol From Scan For Store (store_name) error_code.

Explanation:
The rtl_store_symbol_get_next function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0122E  Error Ending Symbol Scan For Store (store_name) error_code.

Explanation:
The rtl_store_symbol_end_scan function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0123E  Error Adding Symbol (symbol_name) To Store (store_name) error_code.

Explanation:
The rtl_store_symbol_add function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0124E  Error Allocating Flusher For Symbol (symbol_name) In Store (store_name) error_code.

Explanation:
The rtl_store_symbolAllocate_flusher function of the storage system returned an error to the DSE server.
DSEI0125E  DSEI013F

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0125E  Error Deleting Symbol (symbol_name) From Store (store_name) error_code.

Explanation:
The rtl_store_symbol_delete function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0126E  Error Beginning Flush For Store (store_name) error_code.

Explanation:
The rtl_store_flush_begin function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0127E  Error Ending Flush For Store (store_name) error_code.

Explanation:
The rtl_store_flush_end function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0128E  Error opening source for store store_id flusher_id: error_code.

Explanation:
The rtl_transport_source_open function of the transport returned an error to the DSE server. store_id and flusher_id identify the flusher thread in which the error occurred.

User response:
See the log file for a preceding message from the transport for more information.

DSEI0129E  Error closing source for store store_id flusher_id: error_code.

Explanation:
The rtl_transport_source_close function of the transport returned an error to the DSE server. store_id and flusher_id identify the flusher thread in which the error occurred.

User response:
See the log file for a preceding message from the transport for more information.

DSEI0130E  Error Beginning Flush Batch For Store (store_name) error_code.

Explanation:
The rtl_store_flush_batch_begin function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0131E  Error Ending Flush Batch For Store (store_name) error_code.

Explanation:
The rtl_store_flush_batch_end function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0132E  Error Flushing Ticks to Store (store_name) error_code.

Explanation:
The rtl_store_flush_tick function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0133F  Error Opening Storage System Library (storagesystem_name) (library_path) : error_info.

Explanation:
The dynamic library for the specified storage system cannot be opened. error_info provides information on the type of error that was encountered.

User response:
Ensure that the STORESYSXX_LIB_PATH parameter definition for the storage system library is correct. If the value of this parameter is a relative path, the library must be located in a directory that is included in the list of directories specified by the LD_LIBRARY_PATH environment variable.

DSEI0134F  Could Not Resolve Symbol (subsystem) (library_symbol) : error_info.

Explanation:
The specified symbol in the dynamic library for a subsystem (feed handler, storage system, or high availability communications library) cannot be found in the dynamic library. error_info provides information on the type of error that was encountered.

User response:
Ensure that all STORESYSXX_LIB_PATH, HANDLERXX_LIB_PATH, and HA_LIB_PATH
parameters refer to valid dynamic libraries. Ensure that the dynamic libraries are readable by DSE. Ensure that all required library symbols are defined and made externally visible.

DSEI0135E  Error Initializing Storage System  
(storagesystem_name): error_code.

Explanation:  
The rtl_storesys_init function of the storage system returned an error to the DSE server.

User response:  
See the log file for a preceding message from the storage system for more information.

DSEI0136E  Error Shutting Down Storage System  
(storagesystem_name): error_code.

Explanation:  
The rtl_storesys_shutdown function of the storage system returned an error to the DSE server.

User response:  
See the log file for a preceding message from the storage system for more information.

DSEI0137E  Error Closing Storage System Library  
(storagesystem_name)(library_path): error_info.

Explanation:  
The dynamic library for the specified storage system cannot be closed. error_info provides information on the type of error that was encountered.

User response:  
Contact IBM Software Support.

DSEI0138E  Error Opening Storage System  
(storagesystem_name): error_code.

Explanation:  
The rtl_storesys_open function of the storage system returned an error to the DSE server.

User response:  
See the log file for a preceding message from the storage system for more information.

DSEI0139E  Error Closing Storage System  
(storagesystem_name): error_code.

Explanation:  
The rtl_storesys_close function of the storage system returned an error to the DSE server.

User response:  
See the log file for a preceding message from the storage system for more information.

DSEI0140E  Error Beginning Tick Scan For Store  
(store_name) error_code.

Explanation:  
The rtl_store_tick_begin_scan function of the storage system returned an error to the DSE server.

User response:  
See the log file for a preceding message from the storage system for more information.

DSEI0141E  Error Getting Next Tick From Scan For Store  
(store_name) error_code.

Explanation:  
The rtl_store_tick_get_next function of the storage system returned an error to the DSE server.

User response:  
See the log file for a preceding message from the storage system for more information.

DSEI0142E  Error Ending Tick Scan For Store  
(store_name) error_code.

Explanation:  
The rtl_store_tick_end_scan function of the storage system returned an error to the DSE server.

User response:  
See the log file for a preceding message from the storage system for more information.

DSEI0143E  Error Getting Tick Elem For Store  
(store_name) error_code.

Explanation:  
The rtl_store_tick_get_elem function of the storage system returned an error to the DSE server.

User response:  
See the log file for a preceding message from the storage system for more information.

DSEI0144E  Encrypted passwords must be  
number_characters hexadecimal chars long.

Explanation:  
The character length of the encrypted password strings for the GROUPXX_PASSWORD and STORESYSXX_PASSWORD parameters must be equal to number_chars.

User response:  
Use the rtl_make_pass utility to generate encrypted passwords and copy the encrypted string into the configuration file.
DSEI0145E  Passwords must only contain hexadecimal characters.

Explanation:
The encrypted password strings for the GROUPXX_PASSWORD and STORESYSXX_PASSWORD parameters can contain only the characters 0 to 9, a to f, and A to F.

User response:
Use the rtl_make_pass utility to generate encrypted passwords and copy the encrypted string into the configuration file.

DSEI0146E  Parameter parameter_name is defined but other required related parameters are missing.

Explanation:
Certain parameters can be grouped into sets. For example, a feed handler is defined by these parameters:

- HANDLERXX_NAME
- HANDLERXX_LIB_PATH
- HANDLERXX_MSG_PATH
- HANDLERXX_INFO

For any given set, all required parameters must be defined.

User response:
Define the missing parameters for each set that lacks required parameters and restart DSE.

DSEI0147E  QUERY_THREADS [number_threads] must be <= MAX_QUERY_CONNECTS [number_connects].

Explanation:
The number of query threads must be less than or equal to the number of query connections.

User response:
Decrease the value of the QUERY_THREADS parameter or increase the value of the MAX_QUERY_CONNECTS parameter, then restart DSE.

DSEI0148F  The MAX_SHMSIZE variable (maximum_bytes) must be >= SHMSIZE variable (bytes).

Explanation:
The MAX_SHMSIZE parameter defines the maximum amount of shared memory that DSE can use. The SHMSIZE parameter defines the initial amount of shared memory that DSE uses when it starts. The value of the MAX_SHMSIZE parameter must be greater than or equal to the value of the SHMSIZE parameter.

User response:
Increase the value of the MAX_SHMSIZE parameter or decrease the value of the SHMSIZE parameter, then restart DSE.

DSEI0149F  Store names must be unique: store_name.

Explanation:
Every defined store must have a unique name.

User response:
Ensure that all store names that are defined by the STOREXX_NAME parameter are unique.

DSEI0150E  Cannot get message format info for feed feed_id: error_code.

Explanation:
The rtl_feed_get_message_fmts function of the feed handler returned an error to the DSE server.

User response:
See the log file for a preceding message from the feed for more information. Ensure that the feed is configured correctly. If you implemented the feed, check that the rtl_feed_get_message_fmts function is implemented correctly.

DSEI0151E  Could not get service name "service_name".

Explanation:
The specified service name was not found in the /etc/services file.

User response:
Ensure that /etc/services contains an entry that matches the value of the RTL_SERVICE_NAME parameter. If the value of the RTL_SERVICE_NAME parameter contains no digits, DSE assumes that this value is a port number and an entry in /etc/services is not required.

DSEI0152E  Internal error: Connection id is wrong (actual_id != expected_id, socket_id).

Explanation:
A client API connection has an unexpected connection ID.

User response:
Disconnect the client and reconnect it to DSE.

DSEI0153E  Internal error: Could not find rdesc for cid connection_id.

Explanation:
The connection descriptor cannot be found for a client API connection.

User response:
Disconnect the client and reconnect it to DSE.
DSEI0154E  Internal error: Cid not in range: connection_id.

Explanation:
The query connection ID was outside of the range of supported values. A client is attempting to connect with an invalid connection id.

User response:
Disconnect the client with the bad connection ID and reconnect it to DSE. If the error persists, contact the IBM Software Support.

DSEI0155F  Flush period for store store_id must be less than number_seconds seconds.

Explanation:
DSE uses a rolling window of time for storing ticks in shared memory. As the rolling window of time progresses, DSE frees the ticks that are no longer within the window so that new ticks from the feeds can be stored. Ticks cannot be freed from memory until they are flushed to the store.

DSE flushes ticks from memory to the store periodically. The rate of flushing is determined by STOREXX_FLUSH_PERIOD parameter. The value of this parameter must be less than the size of the rolling window of time, as configured by the NUM_INTERVALS and REFRESH_INTERVAL parameters, so that DSE can keep the rolling window moving forward efficiently.

User response:
Try one of the following actions:

- Reduce the value of the STOREXX_FLUSH_PERIOD parameter so that it is less than number_seconds.
- Increase the size of the rolling window of time by adjusting the values of the NUM_INTERVALS and REFRESH_INTERVAL parameters.

DSEI0156E  Failed to init thread_name condition variable. Errno=errno (errno_text).

Explanation:
The condition variable for the specified thread cannot be initialized. errno indicates the error number that was returned by the pthread_cond_init operating system call. errno_text is a text string describing errno.

User response:
Restart DSE. If the error persists, contact IBM Software Support.

DSEI0157E  Failed to init thread_name mutex. Errno=errno (errno_text).

Explanation:
The mutex for the specified thread cannot be initialized. errno indicates the error number that was returned by the pthread_mutex_init operating system call. errno_text is a text string describing errno.

User response:
Restart DSE. If the error persists, contact IBM Software Support.
DSE10162F  Not enough shared memory to allocate headers, either reduce headers or configure more memory.

User response:
To configure fewer headers (symbols), decrease the value of the SECHDR_SIZE parameter.
To configure more memory, increase the value of the SHMSIZE, MAX_SHMSIZE, or ADD_SHMSIZE parameter.

DSE10163I  Allocated room for number_symbols symbols.

Explanation:
DSE logs this message when it initializes shared memory at startup. This message indicates the amount of space in the security header (symbol) pool.

DSE10164E  Bad schema (record type record_id):
Could not parse line: schema_definition.

Explanation:
The RECORDXX_SCHEMA and RECORDXX_META_SCHEMA parameter definitions must begin with a left parenthesis and end with a right parenthesis. All fields in a schema definition must be separated by commas. At least one field must be defined.

User response:
Correct the schema definition and restart DSE.

DSE10165E  Bad schema (record type record_id, field field_number): Name of field field_name longer than maximum_bytes bytes.

Explanation:
The length of a field name in a RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition must be less than the value of maximum_bytes.

User response:
Correct the schema definition and restart DSE.

DSE10166E  Bad schema (record type record_id, field field_number): Column name longer than maximum_bytes bytes.

Explanation:
The length of the optional store column name that is associated with a RECORDXX_META_SCHEMA field name must be less than the value of maximum_bytes.

User response:
Correct the schema definition and restart DSE.

DSE10167E  Bad schema (record type record_id, field field_number): boolean default value must be true or false.

Explanation:
The default value for a BOOLEAN field must be the word true or false. The value is not case sensitive.

User response:
Correct the schema definition and restart DSE.

DSE10168E  Bad schema (record type record_id, field field_number): Invalid field name field_name.

Explanation:
A field name defined by the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter cannot begin with any of the following characters:
- A number.
- A comma.
- A left parenthesis.
- A right parenthesis.
- An unbalanced quotation mark. For example, an open quote for which there is no closing quote, or vice versa.

User response:
Correct the schema definition and restart DSE.

DSE10169E  Bad schema (record type record_id, field field_number): Expected a number for precision.

Explanation:
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, the FLOAT data type takes an optional precision qualifier of the form "(n)" where n is a positive integer.

User response:
Correct the schema definition and restart DSE.

DSE10170E  Bad schema (record type record_id, field field_number): Precision value number is outside the valid range of 1 and maximum_float_precision.

Explanation:
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, the precision qualifier for a FLOAT data type must be an integer between 1 and the value of maximum_float_precision, inclusive.

User response:
Correct the schema definition and restart DSE.
**DSEI0171E**  Bad schema (record type record_id, field field_number): Expected a closing parenthesis after precision value.

**Explanation:**
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, the FLOAT data type takes an optional precision qualifier of the form "(n)" where n is a positive integer.

**User response:**
Correct the schema definition and restart DSE.

**DSEI0172E**  Bad schema (record type record_id): First field must be of type 'timestamp' or 'timestamp(rtl_datetime)'

**Explanation:**
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, the first field must be of one of the following types:
- timestamp
- timestamp(rtl_datetime)

**User response:**
Correct the schema definition and restart DSE.

**DSEI0173E**  Bad schema (record type record_id, field field_number): Keyword STORE only allowed in metadata record definition.

**Explanation:**
The STORE clause is valid only in a RECORDXX_META_SCHEMA definition.

**User response:**
Correct the schema definition and restart DSE.

**DSEI0174E**  Bad schema (record type record_id, field field_number): Default value too large, only maximum_characters characters supported.

**Explanation:**
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, the default value for a CHAR or BYTE field can be at most maximum_characters in length. Note that for a BYTE field, the length in bytes is one half of its hexadecimal string representation.

**User response:**
Correct the schema definition and restart DSE.

**DSEI0175E**  Bad schema (record type record_id, field field_number): Expected a number in default value.

**Explanation:**
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, the keyword DEFAULT must be followed by a value that is appropriate for the field type.

**User response:**
Correct the schema definition and restart DSE.

**DSEI0176E**  Bad schema (record type record_id, field field_number): Default value does not fit in type.

**Explanation:**
The default value for each numeric field in the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition must be within the supported range for that data type. The following list shows the range of values for each numeric data type:

- smallint: -32767 to 32768
- integer: -2147483648 to 2147483648
- bigint: -9223372036854775808 to 9223372036854775808
- real: -3.40282e+38 to 3.40282e+38
- double: -1.79769e+308 to 1.79769e+308

**User response:**
Correct the schema definition and restart DSE.

**DSEI0177E**  Bad schema (record type record_id): Unbalanced parentheses or quotes.

**Explanation:**
The RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition is not enclosed in parentheses or quotation marks, or one of the parentheses or quotation marks is missing.

**User response:**
Ensure that the schema definition is enclosed in parentheses. If any field names or default character values are quoted strings, ensure that each opening quotation mark has a matching closing quotation mark. If any field names contain special characters (such as punctuation or spaces), those characters must be properly escaped with backslashes.

**DSEI0178E**  Bad schema (record type record_id, field field_number): Unexpected keyword or value found: text.

**Explanation:**
While parsing the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, DSE encountered a numeric value or a keyword (WITH, NOT, STORE, NULL, and so on) that is out of place.

**User response:**
Correct the schema definition and restart DSE.
DSEI0179E  Bad schema (record type record_id, field field_number): Current date must be used with a date type.

Explanation:
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, CURRENT DATE is the only allowable default value for a DATE type.

User response:
Correct the schema definition and restart DSE.

---

DSEI0180E  Bad schema (record type record_id, field field_number): Current timestamp must be used with a timestamp type.

Explanation:
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, CURRENT TIMESTAMP is the only allowable default value for a timestamp type. The timestamp types are:
- timestamp(rl_datetime)
- timestamp(rl_java_date)
- timestamp(time_t)
- time_t

User response:
Correct the schema definition and restart DSE.

---

DSEI0181E  Bad schema (record type record_id, field field_number): Current time must be used with a time type.

Explanation:
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, CURRENT TIME is the only allowable default value for a TIME type.

User response:
Correct the schema definition and restart DSE.

---

DSEI0182E  Bad schema (record type record_id, field field_number): Single quotes need to surround field_type default value.

Explanation:
In the RECORDXX_SCHEMA or RECORDXX_META_SCHEMA parameter definition, default values for CHAR and BYTE types must be enclosed in single quotation marks.

User response:
Correct the schema definition and restart DSE.
<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSEI0187E</td>
<td>Error opening target for feed feed_id: error_code.</td>
<td>Explanation: The rtl_transport_target_open function of the transport returned an error to the DSE server. User response: See the log file for a preceding message from the transport for more information.</td>
</tr>
<tr>
<td>DSEI0188E</td>
<td>Error closing target for Feed feed_id: error_code.</td>
<td>Explanation: The rtl_transport_target_close function of the transport returned an error to the DSE server. User response: See the log file for a preceding message from the transport for more information.</td>
</tr>
<tr>
<td>DSEI0189E</td>
<td>Error writing message to target for feed feed_id: error_code.</td>
<td>Explanation: The rtl_transport_target_put_message function of the transport returned an error to the DSE server. User response: See the log file for a preceding message from the transport for more information.</td>
</tr>
<tr>
<td>DSEI0190E</td>
<td>Cannot initialize transport handler (handler_name): error_code.</td>
<td>Explanation: The rtl_transport_handler_init function of the feed handler returned an error to the DSE server. User response: See the log file for a preceding message from the transport for more information. Ensure that the transport is configured correctly. If you implemented the transport, check that the rtl_transport_handler_init function is implemented correctly.</td>
</tr>
<tr>
<td>DSEI0191E</td>
<td>Cannot shut down transport handler (handler_name): error_code.</td>
<td>Explanation: The rtl_transport_handler_shutdown function of the transport returned an error to the DSE server. User response: See the log file for a preceding message from the transport for more information.</td>
</tr>
<tr>
<td>DSEI0192I</td>
<td>Execution begins.</td>
<td>Explanation: DSE logs this message after it initializes shared memory and creates all its threads.</td>
</tr>
<tr>
<td>DSEI0193I</td>
<td>number_ticks ticks added.</td>
<td>Explanation: When the percentage utilization of memory that contains ticks exceeds the value of the PURGE_HWM parameter, DSE adds shared memory segments to accommodate the incoming ticks. This message is logged as each new shared memory segment is allocated and initialized. This message indicates how many ticks can be held in the newly added segment.</td>
</tr>
<tr>
<td>DSEI0194E</td>
<td>Error deleting message from source for thread_name thread_number: error_code.</td>
<td>Explanation: The rtl_transport_source_delete_message function of the transport returned an error to the DSE server. thread_name and thread_number identify the DSE thread that called the transport library function. User response: See the log file for a preceding message from the transport for more information.</td>
</tr>
<tr>
<td>DSEI0195E</td>
<td>Error beginning transport source batch for thread_name thread_number: error_code.</td>
<td>Explanation: The rtl_transport_source_batch_begin function of the transport returned an error to the DSE server. thread_name and thread_number identify the DSE thread that called the transport library function. User response: See the log file for a preceding message from the transport for more information.</td>
</tr>
<tr>
<td>DSEI0196E</td>
<td>Error ending transport source batch for thread_name thread_number: error_code.</td>
<td>Explanation: The rtl_transport_source_batch_end function of the transport returned an error to the DSE server. thread_name and thread_number identify the DSE thread that called the transport library function. User response: See the log file for a preceding message from the transport for more information.</td>
</tr>
<tr>
<td>DSEI0197E</td>
<td>Config file parameter names must begin with an uppercase letter [text_of_line].</td>
<td>Explanation: Configuration file parameter names must begin in uppercase. Some parameter names also contain digits;</td>
</tr>
</tbody>
</table>
### DSEI0198E • DSEI0207E

The digit placeholders are represented as XX in Chapter 9, “System configuration parameters,” on page 51.

**User response:**
Correct the configuration file and restart DSE.

---

**DSEI0198E** 
Config file parameter name is too long or contains illegal characters [text_of_line].

**Explanation:**
Configuration file parameter names can contain only uppercase letters, digits, and underscores. The parameter names must be less than 200 characters in length.

**User response:**
Correct the configuration file and restart DSE.

---

**DSEI0199E** 
Illegal character found after config file parameter name [text_of_line].

**Explanation:**
The only characters that are allowed between parameter names and values are:
- Space
- Tab
- Equal sign

**User response:**
Correct the configuration file and restart DSE.

---

**DSEI0200E** 
Config file parameter value is too long or contains illegal characters [text_of_line].

**Explanation:**
A parameter value can be at most 8192 characters long. A parameter value that is not enclosed in quotation marks can contain only alphanumeric characters and underscores.

**User response:**
Correct the configuration file and restart DSE.

---

**DSEI0201E** 
Config file parameter value not found after name [parameter_name].

**Explanation:**
A value for this parameter must be specified. This parameter does not have a default value.

**User response:**
Specify a value for this parameter and restart DSE.

---

**DSEI0202E** 
Config file parameter value is missing a closing quote [text_of_line].

**Explanation:**
Parameter values that are enclosed in quotation marks must have both beginning and ending quotation marks.

**User response:**
Correct the configuration file and restart DSE.

---

**DSEI0203E** 
Unrecognized config file parameter [parameter_name].

**Explanation:**
The configuration file contains an unknown parameter.

**User response:**
Correct the configuration file and restart DSE.

---

**DSEI0204E** 
Value for parameter parameter_name must be a quoted string.

**Explanation:**
The specified parameter requires a non-numeric value. All non-numeric values must be enclosed in quotation marks.

**User response:**
Ensure that the value is enclosed in quotation marks.

---

**DSEI0205E** 
Config file parameter name must include an n-digit number [parameter_name].

**Explanation:**
Many configuration file parameter names contain one or two digits (for example, STORE01_NAME, FEED23_SOURCE, and UDP7). The digit placeholders are represented by XX in Chapter 9, “System configuration parameters,” on page 51.

**User response:**
Correct the configuration file and restart DSE.

---

**DSEI0206E** 
Error beginning metawrite for store (store_name) error_code.

**Explanation:**
The rtl_store_sym_update_begin function of the storage system returned an error to the DSE server. As a result, symbol metadata was not persisted to the store.

**User response:**
See the log file for a preceding message from the storage system for more information.

---

**DSEI0207E** 
Error ending metawrite for store (store_name) error_code.

**Explanation:**
The rtl_store_sym_update_end function of the storage system returned an error to the DSE server.

**User response:**
See the log file for a preceding message from the storage system for more information.
DSEI0208E ✷ DSEI0216W

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSEI0208E</td>
<td>Error beginning metawrite batch for store (store_name) error_code.</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td>The rtl_store_sym_update_batch_begin function of the storage system returned an error to the DSE server.</td>
</tr>
<tr>
<td><strong>User response:</strong></td>
<td>See the log file for a preceding message from the storage system for more information.</td>
</tr>
<tr>
<td>DSEI0209E</td>
<td>Error ending metawrite batch for store (store_name) error_code.</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td>The rtl_store_sym_update_batch_end function of store store_name returned an error to the DSE server.</td>
</tr>
<tr>
<td><strong>User response:</strong></td>
<td>See the log file for a preceding message from the storage system for more information.</td>
</tr>
<tr>
<td>DSEI0210E</td>
<td>Error getting nulls for symbol metadata (store_name).</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td>Use this message in conjunction with the DSEI0209E message to determine the incorrect store and record type configuration.</td>
</tr>
<tr>
<td><strong>User response:</strong></td>
<td>Ensure that the STOREXX_RECTYPE parameter definition for the specified store corresponds to a RECORDXX_META_SCHEMA parameter definition.</td>
</tr>
<tr>
<td>DSEI0211E</td>
<td>Error writing metadata for symbol symbol_name to store (store_name) error_code.</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td>The rtl_store_sym_update function of store store_name returned an error to the DSE server.</td>
</tr>
<tr>
<td><strong>User response:</strong></td>
<td>See the log file for a preceding message from the storage system for more information.</td>
</tr>
<tr>
<td>DSEI0212E</td>
<td>Value for parameter parameter_name cannot exceed maximum_characters characters.</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td>The string value of the parameter is too long.</td>
</tr>
<tr>
<td><strong>User response:</strong></td>
<td>Reduce the length of the string value and restart DSE.</td>
</tr>
<tr>
<td>DSEI0213E</td>
<td>Transport transport_id: Non-restartable transport cannot be use in restartable mode.</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td></td>
</tr>
</tbody>
</table>

When restartable mode is enabled, all transports that are defined in the DSE configuration file must be restartable. Thus, in the event of a failure, messages from the source can be re-read after a restart so that no data is lost. A transport for socket-based data sources is generally not restartable; a transport for WebSphere MQ queues is restartable.

**User response:**
Check that all transports that are defined in the configuration file are of the appropriate type. If you are using a custom-built transport, ensure that it can support restartability and that it returns RTL_TRANSACTIONAL when its restartability property is queried through the rtl_transport_get_property function.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSEI0214W</td>
<td>run_type metawrite run incomplete: number_commits symbol updates committed, number_discards discarded and number_rollback rolled back for store (store_name).</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td>A manual or automatic writing of metadata to the store completed, but not all of the metadata changes were written to the store.</td>
</tr>
<tr>
<td><strong>User response:</strong></td>
<td>See the log file for a preceding message from the storage system for more information.</td>
</tr>
<tr>
<td>DSEI0215I</td>
<td>run_type metawrite run finished, number_symbols symbols updated and number_discards symbols failed to be updated for Store (store_name).</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td>A manual or automatic writing of metadata to the store completed. number_symbols indicates the number of successful metadata updates. number_discards indicates the number of failed metadata updates.</td>
</tr>
<tr>
<td>DSEI0216W</td>
<td>run_type metawrite run not attempted, Store (store_name) is offline.</td>
</tr>
<tr>
<td><strong>Explanation:</strong></td>
<td>One of the following events occurred:</td>
</tr>
<tr>
<td><strong>User response:</strong></td>
<td>Bring the specified store back online, or see the log file for a preceding message from the storage system to determine why the store is offline.</td>
</tr>
</tbody>
</table>
This message can result from a temporary network failure or a configuration problem in a high availability cluster. Ensure that all of the DSE instances in the cluster use the same version of the software and that the high availability configuration parameters are set to the same values in configuration files. If you have a custom high availability library, contact the author of the library for support.

DSEI0228I  Synchronized metadata for store store_name.

Explanation:
A standby DSE instance in a high availability cluster logs this message after all of the symbol names are synchronized for all of the stores.
A standby DSE instance in a high availability cluster logs this message after all of the symbol metadata is synchronized for each store.

**DSEI0229I** Synchronized metadata in all stores.

**Explanation:**
A standby DSE instance in a high availability cluster logs this message after all of the symbol metadata is synchronized for all of the stores.

**DSEI0230E** Error: Attempt to change state from state1 to state2.

**Explanation:**
An attempt was made to change the state of a DSE instance in a high availability cluster, using the rt1node command. The state change from state1 to state2 is not allowed. state1 is the current state of the DSE instance and state2 is the requested state.

**User response:**
Run rt1stat -h command to determine the state of all of the DSE instances in the cluster.

**DSEI0231E** Error in synchronization, message type type_number: too many symbols in message (number_symbols).

**Explanation:**
The DSE instance in a high availability cluster received an incorrect synchronization message from a peer DSE instance. The response contained more symbols than the number of symbols that were requested.

**DSEI0232E** Error in synchronization, message type type_number: not enough symbols in message.

**Explanation:**
The DSE instance in a high availability cluster received a incorrect synchronization message from a peer DSE instance. The response contained fewer symbols than the number of symbols that were requested.

**User response:**
This message can result from a temporary network failure or a configuration problem in a high availability cluster. Ensure that all of the DSE instances in the cluster use the same version of the software and that the high availability configuration parameters are set to the same values in configuration files. If you have a custom high availability library, contact the author of the library for support.

**DSEI0234I** Symbol symbol_name not returned by peer, reason reason_code.

**Explanation:**
The databases in a high availability cluster are not synchronized, and the DSE instances in the cluster do not share a database. The database used by the standby DSE instance contains the symbol symbol_name; the database used by the live DSE instance does not.

**DSEI0237I** Data source "source_name" opened by feed feed_id.

**Explanation:**
The specified data source was opened by the DSE server. feed_id identifies the DSE thread that called the transport library rtl_transport_source_open function.
DSEI0238I • DSEI0252I

DSEI0238I Data source "source_name" closed by feed feed_id.

Explanation:
The specified data source was closed by the DSE server. feed_id identifies the DSE thread that called the transport library rtl_transport_source_close function.

DSEI0239I Data target "target_name" opened by thread_name thread_number.

Explanation:
The specified data target was opened by the DSE server. thread_name and thread_number identify the DSE thread that called the transport library rtl_transport_target_open function.

DSEI0240I Data target "target_name" closed by thread_name thread_number.

Explanation:
The specified data target was closed by the DSE server. thread_name and thread_number identify the DSE thread that called the transport library rtl_transport_target_close function.

DSEI0241E Error Getting Symbol id From Store (store_name) for symbol symbol_name error_code.

Explanation:
The rtl_store_symbol_getid function of the storage system returned an error to the DSE server.

User response:
See the log file for a preceding message from the storage system for more information.

DSEI0242E Feed feed_id error: Store id mismatch in multi-tick message (expected=expected_store_id, actual=actual_store_id)

Explanation:
In restartable mode, all ticks in a multi-tick message must be assigned to the same store ID. This requirement ensures transactional integrity between the message queue delete operation and the store insert operation.

User response:
In per-symbol partitioning mode, check that all ticks in a multi-tick message have the same symbol ID. In per-tick partitioning mode, check that all ticks in a multi-tick message have the same partition key.

DSEI0243I Initialization begins.

Explanation:
The DSE logs this message when it finishes reading the configuration file and before it initializes shared memory.

DSEI0244E Maximum number of INCLUDEd config files max_number exceeded.

Explanation:
INCLUDE directives in DSE configuration files can reference at most max_number of unique file names.

User response:
Reduce the number of INCLUDE directives by combining one or more configuration sub-files.

DSEI0245E Recursive config file INCLUDEs are not allowed.

Explanation:
DSE detected a configuration file INCLUDE directive that references itself or a previously INCLUDEd configuration file.

User response:
Avoid using INCLUDE directives in configuration files that use INCLUDE directives to include additional files.

DSEI0246E Config file read error.
expected=expected_byte_count
actual=actual_byte_count errno=errno (errno_text).

Explanation:
A read error occurred while saving the contents of the configuration file in shared memory. errno contains the error number that was returned by the read operating system call and errno_text is a text string describing errno.

User response:
Ensure that the configuration file exists and is readable.

DSEI0251F Error Opening High Availability System Library (library_path) : error_info.

Explanation:
The high availability communications library that is specified by the HA_LIB_PATH parameter cannot be found.

User response:
Ensure that the following conditions are met:
• The high availability communications library exists.
• The HA_LIB_PATH parameter definition is correct in the configuration file.
• You have execute permission on the configuration file.

DSEI0252I Data source "source_name" opened by store store_id flusher flusher_id.

Explanation:
The specified data source was opened by the DSE server. store_id and flusher_id identify the DSE thread that called the transport library rtl_transport_source_open function.
**DSEI0253I • DSEI0264E**

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Explanation</th>
<th>User response</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSEI0253I</td>
<td>Data source &quot;source_name&quot; closed by store store_id flusher flusher_id.</td>
<td>The specified data source was closed by the DSE server. store_id and flusher_id identify the DSE thread that called the transport library rtl_transport_source_close function.</td>
<td></td>
</tr>
<tr>
<td>DSEI0254I</td>
<td>HA Standby Handler thread thread_id connected to peer peer_name.</td>
<td>The DSE process that was started in standby mode connected to the specified peer DSE process.</td>
<td></td>
</tr>
<tr>
<td>DSEI0256I</td>
<td>HA system shutdown.</td>
<td>The high availability communications library shut down.</td>
<td></td>
</tr>
<tr>
<td>DSEI0257E</td>
<td>Received unknown initial handshake message from peer peer_name.</td>
<td>The DSE instance in a high availability cluster received an incorrect synchronization message from a peer DSE instance.</td>
<td></td>
</tr>
<tr>
<td>DSEI0258E</td>
<td>Peer peer_name received unknown initial handshake message from a peer.</td>
<td>This message can result from a temporary network failure or a configuration problem in a high availability cluster. Ensure that all of the DSE instances in the cluster use the same version of the software and that the high availability configuration parameters are set to the same values in configuration files. If you have a custom high availability library, contact the author of the library for support.</td>
<td></td>
</tr>
<tr>
<td>DSEI0260E</td>
<td>Unknown thread type type_number at source_code_file line line_number.</td>
<td>An internal error occurred.</td>
<td></td>
</tr>
<tr>
<td>DSEI0261E</td>
<td>Invalid comm_link at source_code_file line line_number.</td>
<td>A peer communications subroutine in the high availability cluster detected an incorrect comm_link pointer.</td>
<td></td>
</tr>
<tr>
<td>DSEI0262E</td>
<td>Error when sending initial handshake message in peer peer_name.</td>
<td>The DSE instance in a high availability cluster received an incorrect synchronization message from a peer DSE instance.</td>
<td></td>
</tr>
<tr>
<td>DSEI0263E</td>
<td>Duplicate HA peer entry peer_name for HAXX_PEER_NAME and HAXX_PEER_NAME.</td>
<td>Two HAXX_PEER_NAME parameter entries in the configuration file have the same values for XX. Every HAXX_PEER_NAME parameter entry in the configuration file must have a unique number.</td>
<td></td>
</tr>
<tr>
<td>DSEI0264E</td>
<td>Store store_name encountered error (thread = thread_name).</td>
<td>The specified store cannot be accessed. thread_name provides a general indication of the type of operation that was being performed at the time of the error. If the</td>
<td></td>
</tr>
</tbody>
</table>
operation can be retried, DSE attempts the operation again. If the connection to the store is lost, DSE attempts to reconnect to the store. If the error persists, the store is taken offline.

**User response:**
See the log file for a preceding message from the storage system for more information. Ensure that the store is up and running, and issue the `rtlmode -u` command to tell DSE that the store is now available.

---

**DSEI0265I** Store *store_name* recovered from error (thread = thread_name).

**Explanation:**
A connection was reestablished with the specified store, or a failed store operation succeeded after being retried.

---

**DSEI0266E** Store *store_name* is offline due to error (thread = thread_name).

**Explanation:**
One of the following events occurred:
- The specified store cannot be brought online in response to the `rtlmode -u` command.
- The store was taken offline after DSE attempted to reconnect to it or after DSE retried a failed operation.

**User response:**
See the log file for a preceding message from the storage system for more information. Ensure that the store is up and running, and issue the `rtlmode -u` command to tell DSE that the store is now available.

---

**DSEI0267I** Store *store_name* is online (thread = thread_name).

**Explanation:**
This message is logged when a store is brought back online in response to the `rtlmode -u` command.

---

**DSEI0268I** Store *store_name* is offline (thread = thread_name).

**Explanation:**
This message is logged when a store is taken offline in response to the `rtlmode -o` command.

---

**DSEI0269I** Signalling thread_name thread thread_id to exit.

**Explanation:**
This message is logged each time a high availability thread is shut down.

---

**DSEI0271I** HA Listener thread accepted connection from peer peer_name.

**Explanation:**
The live DSE instance accepted a connection from a peer DSE instance that is in standby mode.

---

**DSEI0272I** Thread thread_id exiting on termination request.

**Explanation:**
The specified thread will exit.

---

**DSEI0274F** Cannot start with High Availability enabled without any HAXX_PEER_NAME entries.

**Explanation:**
High availability is enabled. However, the configuration file does not contain any HAXX_PEER_NAME parameter entries.

**User response:**
Add HAXX_PEER_NAME parameter entries to the configuration file.

---

**DSEI0275E** Peer index HAXX_PEER_NAME is greater than the maximum allowed value maximum_number.

**Explanation:**
The XX number in the HAXX_PEER_NAME parameter entry in the configuration file is out of the range of supported values. XX can have a value of 00 to 09, inclusive.

**User response:**
Correct the HAXX_PEER_NAME entry in the configuration file.

---

**DSEI0276E** Received invalid peer id peer_id from a new connection.

**Explanation:**
The peer ID number that is supplied in the initial communication between peer DSE instances in a high availability cluster is not recognized.

**User response:**
Ensure that the HAXX_PEER_NAME and HAXX_PEER_INFO parameter entries are the same in the configuration files of all of the DSE instances in the high availability cluster.

---

**DSEI0277E** Received peer id peer_id from a new connection which does not have a HAXX_PEER_NAME entry.

**Explanation:**
The peer ID number is not recognized. It was supplied in the initial communication between peer DSE instances in a high availability cluster.

**User response:**
Ensure that the HAXX_PEER_NAME and HAXX_PEER_INFO parameter entries are the same in the configuration files of all the DSE instances in the high availability cluster.
DSE0278E  Invalid ha_user_data at source_code_file line line_number.

Explanation:
An internal error occurred.

User response:
Contact IBM Software Support.

DSE0279E  Peer index HAXX_PEER_INFO is greater than the maximum allowed value maximum_number.

Explanation:
The XX number in the HAXX_PEER_INFO entry in the configuration file is out of the range of supported values. XX can have a value of 00 to 09, inclusive.

User response:
Correct the HAXX_PEER_INFO parameter entry in the configuration file.

DSE0280E  Unable to terminate thread thread_id, connected to peer peer_name. errno = errno (errno_text).

Explanation:
DSE cannot fully clean up after a communication problem with a peer DSE instance. errno indicates the error number that was returned by the pthread_kill operating system call. errno_text is a text string describing errno.

User response:
This error is temporary and can be ignored. If the error persists, contact IBM Software Support.

DSE0281E  HA library API rtl_ha_init failed. Return status = error_code.

Explanation:
The rtl_ha_init function in the high availability library returned an error.

User response:
If you are using a custom high availability library, investigate the return status with the author of the library. Otherwise, contact IBM Software Support.

DSE0282E  HA library API rtl_hacomm_listen failed. Return status = error_code.

Explanation:
The rtl_hacomm_listen function in the high availability library returned an error.

User response:
If you are using a custom high availability library, investigate the return status with the author of the library. Otherwise, contact IBM Software Support.

DSE0284E  Config parameter parameter_name contains unescaped special character in value [parameter_value].

Explanation:
All special characters (punctuation and spaces) in RECORDXX_SCHEMA and RECORDXX_META_SCHEMA field names must be escaped with backslashes. See Chapter 11, "Rules for delimited identifiers," on page 85 for information on the characters that must be escaped.

User response:
Correct the configuration file and restart DSE.

DSE0285E  Config parameter parameter_name changed from old_value to new_value.

Explanation:
A configuration parameter was set to an unusable value and was adjusted automatically.

User response:
Check the configuration file and change the parameter value to prevent this warning.

DSE0286W  Parameter [parameter_name] is deprecated. The new parameter is [new_parameter_name].

Explanation:
A configuration parameter was used that was deprecated and replaced by a new parameter.

User response:
Check the configuration file and change the parameter to the new parameter to prevent this warning.

DSE0290F  Cannot bind to the RTL_SERVICE_NAME port port_number. errno=errno (errno_text).

Explanation:
The socket for client API connections cannot be initialized. errno indicates the error number that was returned by the bind operating system call. errno_text is a text string describing errno.

User response:
Ensure that the port that is associated with the service name is not already in use. Try a different port.

DSE0291E  Cannot accept connection on RTL_SERVICE_NAME port port_number. errno=errno (errno_text).

Explanation:
DSE cannot accept the socket connection for a client API connection. errno indicates the error number that was returned by the accept operating system call. errno_text is a text string describing errno.

User response:
DSE0294E  •  DSE0304W

Ensure that the port that is associated with the RTL_SERVICE_NAME service name and the next QUERY_THREADS number of ports are not already in use. Try a different port. Try to reconnect the client to DSE.


Explanation:  Depending on operation_type, the trace file could not be opened, written to, flushed, or closed._errno contains the error number that was returned by the operating system call._errno_text is a text string describing_errno.

User response:  Ensure that the user running the rtmloder command has write permissions on the trace directory. Ensure that sufficient disk space is free.

DSE0297E  Invalid trace level level_name for component trace_component_name.

Explanation:  Valid trace level settings are:
•  Off
•  Low
•  Med
•  High

These settings are not case-sensitive.

User response:  Ensure that the trace level for each trace component is set to one of the four supported settings.

DSE0300F  Failed to register atexit handlers for clearing shared memory in use flag.

Explanation:  DSE cannot set up its internal mechanism for ensuring that shared memory is left in a known state if a major failure occurs.

User response:  Ensure that the shared memory key is correct and that no other running DSE instance is using the same key.

Clear the “shared memory in use” flag by running the rtmlode -c command. If that does not fix the problem, delete the shared memory segment by using the UNIX ipcrm command. CAUTION: Running the ipcrm command destroys the shared memory and deletes the diagnostic information that it contains.

DSE0301F  Shared memory segment in use.
  key=shared_memory_key
  id=shared_memory_identifier.

Explanation:  A DSE instance that attempted to start used a shared memory key for a memory segment that was previously used by another instance of DSE. That previous instance of DSE might have terminated in an abnormal state.

User response:  Ensure that the shared memory key is correct and that no other running DSE instance is using the same key.

Restart DSE by running the rtmloder command with the -f command line option.

If that does not fix the problem, delete the shared memory segment by using the UNIX ipcrm command.

CAUTION: Running the ipcrm command destroys the shared memory and deletes the diagnostic information that it contains.

DSE0302E  Duplicate key error adding symbol symbol_name to store store_name.

Explanation:  DSE attempted to add a symbol to a store in which the symbol was already defined. Duplicate symbols are allowed only if high availability is enabled and the store is shared by all of the DSE instances in the high availability cluster.

User response:  Verify that the storage system is properly configured for high availability.

DSE0303F  Another rtmloder is running and using this shared memory.
  key=shared_memory_key
  id=shared_memory_identifier.

Explanation:  A DSE instance that attempted to start used a shared memory key for a memory segment that is currently in use by another running instance of DSE.

User response:  Ensure that the shared memory key is correct and that no other running DSE instance is using the same key.

DSE0304W  Cannot update metadata from tick for the simple schema store store_id (feed feed_id).

Explanation:  DSE received a metadata update message for the specified store from the specified feed. The message cannot be processed because the store uses a simple schema. This warning is logged at most once for each feed.

User response:
Check the feed source to determine why it is sending metadata information to DSE. A store using a simple
schema does not have symbols or metadata information.

DSEI0305E  Unflushable ticks: number_tick ticks not flushed for store store_name flusher flusher_number.

Explanation:
Ticks for the specified store and flusher were not flushed because of storage system errors. The ticks were marked as flushed to avoid subsequent problems and they will be purged at the end of their time window.

User response:
Determine why the ticks were not flushed by examining previous error messages from the storage system in the log.

DSEI0306E  Bad tick free: ticks freed = number_ticks_freed, tick size = tick_size.

Explanation:
An internal error occurred. The ticks cannot be purged.

User response:
Contact IBM Software Support.

DSEI0307F  Failed to initialize store store_id, shutting down

Explanation:
The store with the given store_id and its related storage system could not be initialized.

User response:
Determine why the store and related storage system could not be initialized by examining previous error messages from the store and storage system in the log.

DSEI0313E  Cannot determine the partition mode for store store_name, rc=error_code.

Explanation:
The rtl_store_get_partition_mode function of the store returned an unexpected value.

User response:
See the log file for a preceding message from the store for more information. Ensure that the store is configured correctly.

DSEI0314I  Partition mode for store store_name is "partition_mode".

Explanation:
The given store is using the stated partitioning mode.
Storage system messages

These messages are for the DB2 storage system library that is shipped with DSE.

Other storage systems might have the same error message numbers. You must know which storage system reported the error to interpret it.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Message Description</th>
</tr>
</thead>
</table>
| DSES0001I   | Storage System 
storageystem_name initialized.                                                |
| Explanation:| The specified storage system is successfully initialized.                            |
| DSES0002I   | Storage System 
storageystem_name shutdown.                                                   |
| Explanation:| The specified storage system is successfully shut down.                              |
| DSES0003E   | Memory allocation failed (size bytes at line line_number in file source_code_file). |
| Explanation:| DSE has insufficient virtual memory. line_number and source_code_file indicate the  |
|             | location in the DSE code where the error occurred.                                   |
| User response:| Increase the amount of private virtual memory that is available to a process. Run  |
|             | the UNIX ulimit command to increase the maximum size of the data segment for a user |
| DSES0004E   | Error could not get storesys conn from config (storageystem_name).                  |
| Explanation:| The storage system cannot obtain the value of the STORESYSXX_CONN parameter for the  |
|             | specified storage system. As a result, a connection to the database cannot be        |
|             | established.                                                                          |
| User response:| Ensure that the STORESYSXX_CONN value is correct in the configuration file.          |
| DSES0005E   | Error could not allocate handle_type Handle (store_name).                            |
| Explanation:| DSE cannot allocate a DB2 resource handle.                                            |
| User response:| Ensure that the machine has sufficient resources, especially free memory. Restart   |
|             | DSE. If the error persists, contact IBM Software Support.                              |
| DSES0006E   | Error setting row set size (rowset_size) (store_name).                               |
| Explanation:| The storage system cannot set the row set size attribute in an SQL statement.       |
| User response:| See the log file for a preceding DSES0021E entry that contains SQL error information.|

DSES0007E Error could not get user_data_type user data (store_name).

Explanation: The storage system cannot obtain its user data from the DSE server.

User response: Restart DSE. If the error persists, contact IBM Software Support.

DSES0008E Error unknown store mode (mode_number) (store_name).

Explanation: The rtl_store_open or rtl_store_close function was called with an unknown store mode.

User response: Ensure that you are using the DB2 storage system that is provided with this release of DSE. If the error persists, contact IBM Software Support.

DSES0009E Error could not get store user data (store_name).

Explanation: The storage system cannot obtain its symbol add user data from the DSE server. This user data was created by the rtl_store_symbol_add_begin function.

User response: Ensure that you are using the DB2 storage system that is provided with this release of DSE. Restart DSE. If the error persists, contact IBM Software Support.

DSES0010E Error executing statement_type statement (store_name).

Explanation: The SQLExecute CLI function cannot execute an SQL statement.

User response: See the log file for a preceding DSES0021E entry that contains SQL error information.
DSES0011E  Error executing fetch_type fetch (store_name).

Explanation:
The SQLFetch CLI function cannot get data from the database.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0012E  Error closing cursor_type cursor (store_name).

Explanation:
The SQLCloseCursor CLI function cannot close an insert or fetch cursor.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0013E  Error setting attribute_type attribute (attribute_value) (store_name).

Explanation:
The SQLSetConnectAttr or SQLSetStmtAttr CLI function cannot set the specified attribute.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0014E  Error preparing statement_type statement (statement) (store_name).

Explanation:
The SQLPrepare CLI function cannot prepare an SQL statement.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0015E  Error binding column_name column (statement_type) (store_name).

Explanation:
The SQLBindCol CLI function cannot bind the specified column name to its corresponding application variable for a select statement.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0016E  Error binding column_name parameter (statement_type) (store_name).

Explanation:
The SQLBindParameter CLI function cannot bind the specified column name to its corresponding parameter marker for an insert or update statement.

DSES0017E  Error setting parameter set size (set_size) (store_name).

Explanation:
The SQLSetStmtAttr CLI function cannot set the SQL_ATTR_PARAMSET_SIZE parameter for an insert or update statement.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0018E  Error flushing ticks, attempted number_attempted, actual number_succeeded, (store_name).

Explanation:
The DSE storage system cannot flush all of the ticks to the store_name_ticks table.

User response:
Determine if a problem exists with the database. Check that the data that is supplied by the feed is correct and that the feed handler is passing the data to DSE correctly.

DSES0019E  Error action_type tick flush batch transaction (store_name).

Explanation:
The DSE storage system cannot commit or abort the transaction to flush ticks to the store_name_ticks table.

User response:
Determine if a problem exists with the database. Check that the data that is supplied by the feed is correct and that the feed handler is passing the data to DSE correctly.

DSES0020E  Error unknown batch end mode (mode_number) (store_name).

Explanation:
The DSE server called the rtl_store_flush_batch_end or rtl_store_symbol_add_batch_end function with an unknown batch end mode. The batch end mode must be one of the following modes:
- RTL_STORE_BATCH_COMMIT
- RTL_STORE_BATCH_ABORT

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. If the error persists, contact IBM Software Support.
DSES0021E • DSES0028E

DSES0021E  SQL Error (store_name) SQLSTATE = sql_state Native Error Code = sql_code
Message = sql_message_text.

Explanation:
The storage system called a DB2 CLI function, which returned SQL_ERROR. This message provides the error information that is supplied by the SQLGetDiagRec CLI function.

User response:
Use this information to help determine the cause of a database-related storage system error.

DSES0022E  SQL Error (store_name) check return code got unknown error error_code.

Explanation:
A DB2 CLI function returned SQL_INVALID_HANDLE or an unknown error code.

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. Determine if a problem exists with the database. If the error persists, contact IBM Software Support.

DSES0023E  Error setting attribute_type attribute attribute_name (store_name).

Explanation:
One of the following events occurred:
• The SQLSetEnvAttr function cannot set an attribute in an environment.
• The SQLSetConnectAttr CLI function cannot set a connection handle.

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. Determine if a problem exists with the database. If the error persists, contact IBM Software Support.

DSES0024E  Error connecting to database database_name (store_name).

Explanation:
The SQLConnect CLI function cannot connect to the specified database.

User response:
Ensure that:
• The STORESYSXX_CONN parameter contains the name of the database.
• The STORESYSXX_USER and STORESYSXX_PASSWORD parameters contain the name and password of a user who has permission to connect to the database.

DSES0025E  Error freeing statement resources (resource_type) (store_name).

Explanation:
The SQLFreeStmt CLI function cannot free statement resources.

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. Determine if a problem exists with the database. If the error persists, contact IBM Software Support.

DSES0026E  Error getting record type info (record_type_number) (store_name).

Explanation:
The rtl_get_tick_info or rtl_get_meta_info storage API function returned an error.

User response:
Ensure that record_type_number corresponds to a configured RECORDXX_SCHEMA or RECORDXX_META_SCHEMA definition.

DSES0027W  Record type record_type_name contains no fields which can be persisted to store store_name.

Explanation:
None of the fields in the associated RECORDXX_META_SCHEMA definition have STORE READWRITE or STORE READWRITE AUTO attributes, so no symbol metadata is persisted to the store.

User response:
Ensure that at least one field in the RECORDXX_META_SCHEMA definition for this store has a STORE READWRITE or STORE READWRITE AUTO attribute. Otherwise, set STOREXX_META_MODE to 0 to disable metadata persistence.

DSES0028E  Error unknown field type (field=field_number; type=record_type_number) (store_name).

Explanation:
The storage system encountered an unknown field type during the transfer of data between the database and DSE shared memory.

User response:
Ensure that all of the fields in the associated RECORDXX_SCHEMA or RECORDXX_META_SCHEMA are of data types that are supported by the storage system. For more information, see:
DSES0030E  Error rowset data or memory data is NULL (store_name).

Explanation:
The storage system encountered a null source or destination pointer during the transfer of data between the database and DSE shared memory.

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. Check that the data that is supplied by the feed is correct and that the feed handler is passing the data to DSE correctly. If the error persists, contact IBM Software Support.

DSES0032E  Error getting table partition info (operation) (store_name).

Explanation:
The storage system cannot obtain the partitioning map for the symbol table.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information. Ensure that the user specified by STORESYSX_USER parameter definition has at least one of the following settings for the symbol table:

• sysadm authority
• dbadm authority
• CONTROL privilege
• SELECT privilege

DSES0034E  Flusher flusher_id assigned to node node_id.

Explanation:
The storage system logs this message when a store is opened for flushing ticks. This message indicates that the storage system will connect to the specified DB2 node.

DSES0035E  Error could not get symbol update user data (store_name).

Explanation:
The storage system cannot obtain its symbol update user data from the DSE server. This user data was created by the rtl_store_symbol_update_begin function.

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. Restart DSE. If the error persists, contact IBM Software Support.

DSES0036E  Error writing symbol metadata, attempted number_rows_attempted, actual number_rows_written. (store_name).

Explanation:
The SQLExecute CLI function cannot update symbol metadata in the database. The rows in the rowset were not all successfully processed.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0037E  Error committing batch transaction (store_name).

Explanation:
The SQLEndTran CLI function cannot commit a batch of symbol inserts or updates.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0038E  Error rolling back batch transaction (store_name).

Explanation:
The SQLEndTran CLI function cannot roll back a batch of symbol inserts or updates.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

DSES0039E  Error unknown symbol update batch end mode (node_number) (store_name).

Explanation:
The DSE server called the rtl_store_symbol_update_batch_end function with an unknown batch end mode. The batch end mode must be one of the following modes:

• RTL_STORE_BATCH_COMMIT
• RTL_STORE_BATCH_ABORT

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. If the error persists, contact IBM Software Support.

DSES0040E  Error getting metadata record type info (record_type_number) (store_name).

Explanation:
The rtl_get_meta_info storage API function returned an error.

User response:
Ensure that record_type_number corresponds to a configured RECORDXX_META_SCHEMA parameter definition.
DSES0043E Error could not get storesys user data (store_name).

Explanation:
The storage system cannot obtain its storage system user data from the DSE server. This user data was created by the rtl_storesys_init function.

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. Restart DSE. If the error persists, contact IBM Software Support.

---

DSES0044E Error setting statement attribute (attribute_type) (store_name).

Explanation:
The SQLSetStmtAttr CLI function cannot set the specified statement attribute.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

---

DSES0045E Error setting connection attribute (attribute_type) (store_name).

Explanation:
The SQLSetConnectAttr CLI function cannot set the specified connection attribute.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

---

DSES0046E Error preparing symbol update (Character_set) statement (statement) (store_name).

Explanation:
The SQLPrepare CLI function cannot prepare a symbol update SQL statement.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

---

DSES0047E Error executing symbol update statement (store_name).

Explanation:
The SQLExecute CLI function cannot execute a symbol update SQL statement.

User response:
See the log file for a preceding DSES0021E entry that contains SQL error information.

---

DSES0048E Error could not get symbol add user data (store_name).

Explanation:
The storage system cannot obtain its symbol add user data from the DSE server. This user data was created by the rtl_store_symbol_add_begin function.

User response:
Ensure that you are using the DB2 storage system that is provided with this release of DSE. Restart DSE. If the error persists, contact IBM Software Support.

---

DSES0049E Could not get table names for Store store_id (store_name).

Explanation:
The storage system cannot construct the names of the tick and symbol tables. These table names come from the following tables:

- name_ticks
- name_symbols

where name is the value of the STOREEXX_NAME parameter definition.

User response:
Ensure that the STOREEXX_NAME parameter for the specified store_id is defined.

---

DSES0050W Nullable partition key column column_name is not defined for store store_id.

Explanation:
A column that is used for the stores ticks table partitioning key is not defined for the record type of the DSE store.

User response:
This message is a warning, not an error, but you should check your database table partitioning key and the DSE record type for the store.

---

DSES0051E Non-nullable Partition key column column_name is not defined for store store_id.

Explanation:
A column that is required for the stores ticks table partitioning key is not defined for the record type of the DSE store.

User response:
Check your database table partitioning key and the DSE record type for the store. Ensure that all required partitioning columns have corresponding fields in the store record type schema definition.
**High availability communication library messages**

These messages are for the high availability communications library that is shipped with DSE.

Other high availability communications libraries can have the same error message numbers; therefore you need to know which library reported the error to be able to properly interpret it.

---

**DSES0052E**  
**Explanation:**  
DSE tried to open a store in a mode that is not supported for the schema type.  
**User response:**  
Contact IBM Software Support.

**DSES0053E**  
**Explanation:**  
DSE tried to open a store in a mode that is not supported by the store.  
**User response:**  
Contact IBM Software Support.

---

**DSEH0001I**  
**Explanation:**  
The high availability communication library initialized.

**DSEH0002F**  
**Explanation:**  
DSE cannot allocate memory, because of one of the following reasons:  
- DSE does not have enough space in the data segment.  
- DSE has insufficient virtual memory.

*source_code_file* and *line_number* indicate the location in the DSE code where the error occurred.

**User response:**  
Increase the amount of private virtual memory that is available to a process. Run the UNIX ulimit command to increase the maximum size of the data segment for a user process.

**DSEH0003F**  
**Explanation:**  
The high availability communication library cannot get the network name of the host system.

**User response:**  
Verify that the host system is configured for networking.

**DSEH0005E**  
**Explanation:**  
The listen() system function failed.  
**User response:**  
Verify that the host system is configured for networking.

**DSEH0006E**  
**Explanation:**  
The accept() system function failed.  
**User response:**  
Verify that the host system is configured for networking.

**DSEH0009E**  
**Explanation:**  
The gethostbyname() system function failed.  
**User response:**  
Verify that the host system is configured for networking.

**DSEH0010E**  
**Explanation:**  
The socket() system function failed.  
**User response:**  
Verify that the host system is configured for networking.
include file that is in the /usr/include directory. See the man page for the socket() function to identify the error and for an explanation of the error.

**DSEH0013E** Cannot bind() socket. **Erno = [errno].**

**Explanation:**
The bind() system function failed. errno contains the system error code.

**User response:**
Verify that the host system is configured for networking. Identify the error number in the errno.h include file that is in the /usr/include directory. See the man page for the bind() function to identify the error and for an explanation of the error.

**DSEH0015E** No valid peer entry for **hostname.**

**Explanation:**
The HAXX_PEER_INFO parameter for this host system is not defined in the configuration file.

**User response:**
In the configuration file, define the HAXX_PEER_INFO parameter for this host system.

**DSEH0018E** Invalid peer entry setting **peer_info.**

**Explanation:**
The HAXX_PEER_INFO parameter for this host system is not defined correctly in the configuration file.

**User response:**
In the configuration file, correct the HAXX_PEER_INFO parameter definition for this host system.

**DSEH0019F** Current host machine **hostname does not match any peer entry in configuration.**

**Explanation:**
The HAXX_PEER_INFO parameter for this host system is not defined in the configuration file.

**User response:**
In the configuration file, define the HAXX_PEER_INFO parameter for this host system.

**DSEH0020E** Unable to set socket option **option. Erno = [errno].**

**Explanation:**
The setsockopt() system function failed. errno contains the system error code.

**User response:**
Verify that the host system is configured for networking. Identify the error number in the errno.h include file that is in the /usr/include directory. See the man page for the setsockopt() function to identify the error and for an explanation of the error.

**DSEH0021E** Invalid comm_link at **source_code_file line line_number.**

**Explanation:**
An internal error occurred.

**User response:**
Contact IBM Software Support.

**DSEH0022E** comm_link from host **hostname not connected to any peer.**

**Explanation:**
An internal error occurred.

**User response:**
Contact IBM Software Support.

**DSEH0023E** Error while reading message size from **socket in host hostname. Erno = [errno].**

**Explanation:**
A message from a peer DSE instance in a high availability cluster cannot be read. This error is temporary and can be ignored. The error often occurs when a peer DSE instance disconnects after it completes synchronization.

**User response:**
If this error occurs frequently and prevents synchronization from completing, the error indicates a network reliability problem between the peer DSE instances in the high availability cluster.

**DSEH0024E** Error while reading message of size **number_bytes from socket in host hostname. Erno = [errno].**

**Explanation:**
A message from a peer DSE instance in a high availability cluster cannot be read. This error is temporary and can be ignored.

**User response:**
If this error occurs frequently and prevents synchronization from completing, the error indicates a network reliability problem between the peer DSE instances in the high availability cluster.

**DSEH0025E** Error while writing message size to **socket in host hostname. Erno = [errno].**

**Explanation:**
A message from a peer DSE instance in a high availability cluster cannot be written. This error is usually temporary and can be ignored. If this error occurs frequently and prevents synchronization from completing, the error indicates a network reliability problem between the peer DSE instances in the high availability cluster.
**DSEH0026E • DSET0013E**

**Explanation:**
The configuration file does not contain any entries of the HAXX_PEER_INFO parameter.

**User response:**
Define the HAXX_PEER_INFO parameter in the configuration file.

---

**DSEH0030F HAXX_PEER_NAME peer_name does not have a corresponding HAXX_PEER_INFO config entry.**

**Explanation:**
A HAXX_PEER_NAME parameter definition in the configuration file does not have a corresponding HAXX_PEER_INFO parameter definition.

**User response:**
Define the HAXX_PEER_INFO parameter in the configuration file.

---

**Pre-built transport library messages**

These messages are for the pre-built transport libraries that are shipped with DSE.

**DSET0001I Transport transport_id (transport_name) initialized.**

**Explanation:**
The specified transport is initialized.

**DSET0002I Transport transport_id (transport_name) shut down.**

**Explanation:**
The specified transport was shut down.

**DSET0010F Memory allocation failed (size bytes at line line_number in file source_code_file).**

**Explanation:**
The transport has insufficient virtual memory. `line_number` and `source_code_file` indicate the location in the transport code where the error occurred.

**User response:**
Increase the amount of private virtual memory that is available to a process. Run the UNIX `ulimit` command to increase the maximum size of the data segment for a user process.

**DSET0011E Transport transport_id does not support message format type format_type.**

**Explanation:**
The specified transport does not support the specified message format type.

---

Chapter 30. Messages 479
DSET0014E • DSET0020E

User response:
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0014E Transport transport_id: Bad
source_or_target message format
definition: format_field_name
(format_field_value) must be >= min_value
and <= max_value.

Explanation:
The value of the specified field for the specified transport’s source or target message definition is out of range and must be:
• Greater than or equal to the given min_value.
• Less than or equal to the given max_value.

User response:
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0015E Transport transport_id: Bad
source_or_target message format
definition: Must specify end indicator.

Explanation:
The specified transport’s source or target message definition does not specify a required end indicator.

User response:
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0016E Transport transport_id: Bad
source_or_target message format
definition: Must specify both start and end indicators.

Explanation:
The specified transport’s source or target message definition does not specify required start and end indicators.

User response:
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0017E Transport transport_id: Bad
source_or_target message format
definition: start and end indicators must be different.

Explanation:
The specified transport’s source or target message definition start and end indicators are the same. They must be unique.

User response:
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0018E Transport transport_id: Bad
source_or_target message format
definition: separator and end indicator must be different.

Explanation:
The specified transport’s source or target message definition separator and end indicators are the same. They must be unique.

User response:
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0019E Transport transport_id: Bad
source_or_target message format
definition: Unknown message format type (format_type).

Explanation:
The specified transport’s source or target message definition format type is not recognized as a valid format type.

User response:
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0020E Transport transport_id: message to be published is too short, expecting >= min_value bytes.

Explanation:
The size of the message to be published by the specified transport must be greater than or equal to the given min_value in bytes.

User response:
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct.

Check that the feed is constructing the message to be published correctly in the rtl_feed_get_publish_message function in the feed handler. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0201E** Transport transport_id: value in length indicator (length) too small.

**Explanation:**
The length of the message for the specified transport is too low. It is possible that the length_ind_start and length_ind_incl format parameters in the message format definition are inconsistent with the actual value found in the message length indicator.

**User response:**
Check that the message format definition that is returned by the rtl_feed_get_message_fmts function in the feed handler is correct. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0100E** Transport transport_id: Cannot open data source_or_target "source_or_target_name": error_text.

**Explanation:**
The specified transport cannot open the specified source or target due to the given error.

**User response:**
Check the error text for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0101E** Transport transport_id: File read error: error_text.

**Explanation:**
The specified transport encountered an error while reading from a file.

**User response:**
Check the error text for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0102E** Transport transport_id: File write error: error_text.

**Explanation:**
The specified transport encountered an error while writing to a file.

**User response:**
Check the error text for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0103E** Transport transport_id: Unexpected end of file; num_bytes bytes requested, bytes_available bytes received.

**Explanation:**
The specified transport unexpectedly encountered the end of file and could not read the requested number of bytes from a file.

**User response:**
Check the data file being processed. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0104E** Transport transport_id: Unexpected end of file; bytes_available leftover bytes.

**Explanation:**
The specified transport unexpectedly encountered the end of file and has a number of bytes that remain unprocessed for a file.

**User response:**
Check the data file being processed. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0200E** Transport transport_id: Cannot create socket for data source_or_target "source_or_target_name": error_text.

**Explanation:**
The specified transport cannot create a socket for the specified source or target due to the given error.

**User response:**
Check the error text for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0201E** Transport transport_id: Cannot connect to data source_or_target "source_or_target_name": error_text.

**Explanation:**
The specified transport cannot connect to the specified source or target due to the given error.

**User response:**
Check the error text for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.

**DSET0202E** Transport transport_id: Cannot parse data source_or_target name "source_or_target_name:"

**Explanation:**
DSET0203E • DSET0209E

The specified transport cannot parse the specified source or target name. The source or target name must be in one of the following formats:
• ip_address:port_num (for example, 127.0.0.1:1234)
• ip_address:service_name (for example, 127.0.0.1:xyz)
• host_name:port_num (for example, pequod:1234)
• host_name:service_name (for example, pequod.moby.ibm.com:xyz)

User response:
See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0203E  Transport transport_id: Cannot get hostname for data source_or_target "source_or_target_name."

Explanation:
The specified transport cannot get the host address from the host name for the specified source or target name. The source or target name must be in one of the following formats:
• ip_address:port_num (for example, 127.0.0.1:1234)
• ip_address:service_name (for example, 127.0.0.1:xyz)
• host_name:port_num (for example, pequod:1234)
• host_name:service_name (for example, pequod.moby.ibm.com:xyz)

User response:
See the log file for a related message from the feed and check that the feed is configured correctly. Ensure that the DSE server knows the host name and can resolve it.

DSET0204E  Transport transport_id: Cannot get service "service_name: for data source_or_target "source_or_target_name."

Explanation:
The specified transport cannot get the port number from the service name for the specified source or target name. The source or target name must be in one of the following formats:
• ip_address:port_num (for example, 127.0.0.1:1234)
• ip_address:service_name (for example, 127.0.0.1:xyz)
• host_name:port_num (for example, pequod:1234)
• host_name:service_name (for example, pequod.moby.ibm.com:xyz)

User response:
See the log file for a related message from the feed and check that the feed is configured correctly. Ensure that the DSE server knows the service name and can resolve it.

DSET0205E  Transport transport_id: Socket read error: error_text.

Explanation:
The specified transport encountered an error while reading from a socket.

User response:
Check the error text for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0206E  Transport transport_id: Socket write error: error_text.

Explanation:
The specified transport encountered an error while writing to a socket.

User response:
Check the error text for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0207E  Transport transport_id: Unexpected socket closure; num_bytes bytes requested, bytes_available bytes received.

Explanation:
The specified transport unexpectedly encountered a socket closure and could not read the requested number of bytes from the socket.

User response:
Check the socket that is being processed. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0208E  Transport transport_id: Unexpected socket closure; bytes_available leftover bytes.

Explanation:
The specified transport unexpectedly encountered a socket closure and has a number of bytes that remain unprocessed for the socket.

User response:
Check the socket that is being processed. See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0209E  Transport transport_id: Cannot set option_name option for source_or_target socket: error_text.

Explanation:
The specified transport could not set the given socket option for the source or target socket.

User response:
Check the error text for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.
DSET0300E  Transport transport_id: MQ_function call failed, reason code $mq_rc$ ($mq_rc_text$).

Explanation:
The indicated WebSphere MQ function returned an error to the specified transport. $mq_rc$ is the WebSphere MQ API reason code number and $mq_rc_text$ is the reason code name.

User response:
Check the WebSphere MQ messages information for more details on the error encountered. See the log file for a related message from the feed and check that the feed is configured correctly.

Ensure that your WebSphere MQ queue manager is configured properly; particularly, ensure that the XAResourceManager stanzas in the qm.ini file are correct. Ensure that the DB2 database was started.

DSET0301W  Transport transport_id: MQ_function call warning, reason code $mq_rc$ ($mq_rc_text$).

Explanation:
The indicated WebSphere MQ function returned a warning to the specified transport. $mq_rc$ is the WebSphere MQ API reason code number and $mq_rc_text$ is the reason code name.

DSET0304E  Transport transport_id: Queue manager name length must be $\leq max$_bytes bytes.

Explanation:
The name of the WebSphere MQ queue manager for the specified transport is too long. The queue manager name length must be less than or equal to the given number of bytes.

User response:
See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0305E  Transport transport_id: Queue name length must be $\leq max$_bytes bytes.

Explanation:
The name of the WebSphere MQ queue for the specified transport is too long. The queue name length must be less than or equal to the given number of bytes.

User response:
See the log file for a related message from the feed and check that the feed is configured correctly.

DSET0306E  Transport transport_id: All source and target queues must be managed by the same queue manager.

Explanation:
The queue manager must be the same for all source and target queues on a single feed. A feed thread can connect to only one queue manager.

User response:
Ensure that the queue manager component of the FEEDXX_SOURCE parameter is the same as the queue manager component of the corresponding FEEDXX_TARGET parameter.

Chapter 30. Messages  483
Chapter 31. DSE traces

DSE provides a trace mechanism to aid in troubleshooting problems. Set up the trace file and settings using the trace configuration parameters.

Determine where to save the trace information by specifying a directory for the trace file, using the TRACEFILE_DIR parameter.

The trace file name has the following format:

\{program_name\}.\{pid\}.\{n\}

Where:

- **program name**
  - Is rtloader.
- **pid**
  - Is the process ID of the running program.
- **n**
  - Is the sequence number of the trace file. The first trace file has a sequence number of 0. The second trace file has a sequence number of 1, and so on.

**Example:**

If:

- The process ID of the running DSE program is 12045.
- The trace directory, specified by the TRACEFILE_DIR parameter, is /tmp/myuser.

Then the trace files have the following names:

- /tmp/myuser/rtloader.12045.0
- /tmp/myuser/rtloader.12045.1
- /tmp/myuser/rtloader.12045.2
- ...

If DSE encounters any I/O errors while writing information to the trace file, tracing is turned off.

You can dynamically turn tracing on or off by running the rtlmode -T command.

**Related reference**

- [“Trace configuration parameters” on page 79](#)
  - Use the trace configuration parameters to set up tracing for the components of DSE.
- [“Trace level configuration parameters” on page 80](#)
  - Use the trace level configuration parameters to enable tracing for the components of DSE and specify the trace level for each component. By default, tracing is turned off.
- [“Trace mechanism for feed handlers” on page 159](#)
  - DSE provides a trace mechanism for feed handlers to add trace messages into the DSE trace output, where the DSE system trace messages are logged. This trace output provides a single location for all DSE trace messages and should be used for all feed handler tracing.
DSE provides a mechanism for transport handlers to add trace messages into the DSE trace output file.
Chapter 32. rtlsmdmp utility for troubleshooting

The rtlsmdmp utility enables you to examine internal memory structures to help troubleshoot problems.

You can analyze configuration and runtime information, including program state, context, and data being loaded.

Related reference
“rtlsmdmp utility” on page 40

The rtlsmdmp utility prints the information in DSE shared memory, to help troubleshoot problems.
Part 7. Appendixes
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.
IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.
Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name)(year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

### Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both.

- AIX
- DB2
- DB2 Universal Database
- HACMP
- IBM
- Tivoli
- WebSphere

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft®, Windows, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel® and Pentium® are trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.
# Index

## A

- **ADD_SHMSIZE** configuration parameter 68
- administration
  - configuring DSE 22, 51
  - configuring for high availability 57, 71
  - controlling DSE 32
  - converting bad data 31
  - creating passwords 49
- description 29
- installation verification 17, 20
- installing DSE 9, 11
- monitoring DSE operation 43
- monitoring DSE performance 43
- printing information in DSE shared memory 40
- quick installation verification 19
- running DSE in restartable mode 52
- starting DSE 23, 29
- storage systems 123

## APIs

- C 285
  - description 285, 293
  - feed handler 287
  - functions 285
  - high availability
    - description 288
    - information accessor 288
  - state 288
- installation 11
- Java 13, 289
- public header files 435
- record type 289
- `rtl_feed_source_get_message` 319, 321
- `rtl_add_seconds` 294
- `rtl_api_version` 295
- `rtl_attach` 296
- `rtl_begin_scan` 298
- `rtl_client_log` 300
- `rtl_client_log_done` 301
- `rtl_client_log_init` 302
- `rtl_detach` 303
- `rtl_elem` 304
- `rtl_end_scan` 306
- `rtl_feed_get_gapfill` 307
- `rtl_feed_get_handler_id` 308
- `rtl_feed_get_handler_name` 309
- `rtl_feed_get_info` 310
- `rtl_feed_get_source` 311
- `rtl_feed_get_store_id` 312
- `rtl_feed_get_store_name` 313
- `rtl_feed_get_target` 314
- `rtl_feed_get_transport_id` 315
- `rtl_feed_get_user_data` 316
- `rtl_feed_is_configured` 317
- `rtl_feed_log` 318
- `rtl_feed_source_put_message` 320
- `rtl_get_feed_for_symbol` 323
- `rtl_get_meta_info` 326
- `rtl_get_meta_rec_info` 327
- `rtl_get_metadata` 324
- `rtl_get_next` 328
- `rtl_get_rectype_id_from_name` 330

## APIs (continued)

- `rtl_get_rtltime_date_fields` 331
- `rtl_get_rtltime_fields` 332
- `rtl_get_rtltime_time_fields` 333
- `rtl_get_tick_info` 334
- `rtl_get_tick_rec_info` 335
- `rtl_get_udp_value` 336
- `rtl_gettime_rtltime` 337
- `rtl_has_feed_from_name` 338
- `rtl_has_feed_info_from_id` 339
- `rtl_has_feed_from_id` 340
- `rtl_log` 341
- `rtl_handler_get_feed_from_name` 342
- `rtl_handler_get_info` 343
- `rtl_handler_get_name` 344
- `rtl_handler_get_user_data` 345
- `rtl_handler_is_configured` 346
- `rtl_handler_log` 347
- `rtl_hasys_get_feed` 348
- `rtl_hasys_get_user_data` 349
- `rtl_localtime_to_rtltime` 350
- `rtl_minus_seconds` 351
- `rtl_put_metadata` 352
- `rtl_reattach` 354
- `rtl_rtljavadate_to_rtltime` 356
- `rtl_rtltime_cmp` 357
- `rtl_rtltime_cpy` 358
- `rtl_rtltime_date_to_string` 359
- `rtl_rtltime_to_rtltime` 360
- `rtl_rtltime_to_string` 361
- `rtl_rtltime_to_tm` 362
- `rtl_rtltime_to_tm` 363
- `rtl_rtltime_to_string` 364
- `rtl_set_meta_defaults` 365
- `rtl_set_rtltime_fields` 366
- `rtl_set_tick_defaults` 367
- `rtl_state_change_register` 368
- `rtl_state_get` 369
- `rtl_state_get_info` 370
- `rtl_state_get_info` 371
- `rtl_stats` 372
- `rtl_store_get_flush_batch_size_from_desc` 374
- `rtl_store_get_flush_batch_size_from_id` 375
- `rtl_store_get_id_from_desc` 376
- `rtl_store_get_id_from_name` 377
- `rtl_store_get_info_from_id` 379
- `rtl_store_get_meta_batch_size_from_desc` 380
- `rtl_store_get_meta_batch_size_from_id` 381
- `rtl_store_get_meta_mode_from_desc` 382
- `rtl_store_get_meta_mode_from_id` 383
- `rtl_store_get_meta_mode_from_id` 384
- `rtl_store_get_meta_period_from_desc` 385
- `rtl_store_get_meta_period_from_id` 386
- `rtl_store_get_name_from_desc` 387
- `rtl_store_get_name_from_id` 388
- `rtl_store_get_num_flushers_from_desc` 389
- `rtl_store_get_num_flushers_from_id` 390
- `rtl_store_get_partition_mode_from_desc` 391
- `rtl_store_get_partition_mode_from_id` 392
- `rtl_store_get_rectype_from_desc` 393
- `rtl_store_get_rectype_from_id` 394

© Copyright IBM Corp. 2004, 2007
<table>
<thead>
<tr>
<th>APIs (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl_store_get_schema_name_from_desc 394</td>
</tr>
<tr>
<td>rtl_store_get_schema_name_from_id 395</td>
</tr>
<tr>
<td>rtl_store_get_schema_type_from_desc 396</td>
</tr>
<tr>
<td>rtl_store_get_schema_type_from_id 397</td>
</tr>
<tr>
<td>rtl_store_get_storesys_id_from_desc 398</td>
</tr>
<tr>
<td>rtl_store_get_storesys_user_data_from_desc 399</td>
</tr>
<tr>
<td>rtl_store_get_symbol_set_size_from_desc 400</td>
</tr>
<tr>
<td>rtl_store_get_symbol_set_size_from_id 401</td>
</tr>
<tr>
<td>rtl_store_get_symbol_table_name_from_desc 402</td>
</tr>
<tr>
<td>rtl_store_get_symbol_table_name_from_id 403</td>
</tr>
<tr>
<td>rtl_store_get_sysname_from_desc 404</td>
</tr>
<tr>
<td>rtl_store_get_sysname_from_id 405</td>
</tr>
<tr>
<td>rtl_store_get_tick_set_size_from_desc 406</td>
</tr>
<tr>
<td>rtl_store_get_tick_set_size_from_id 407</td>
</tr>
<tr>
<td>rtl_store_get_tick_table_name_from_desc 408</td>
</tr>
<tr>
<td>rtl_store_get_tick_table_name_from_id 409</td>
</tr>
<tr>
<td>rtl_store_get_user_data_from_desc 410</td>
</tr>
<tr>
<td>rtl_store_is_configured 411</td>
</tr>
<tr>
<td>rtl_storesys_get_conn_from_desc 412</td>
</tr>
<tr>
<td>rtl_storesys_get_conn_from_id 413</td>
</tr>
<tr>
<td>rtl_storesys_get_ha_from_id 414</td>
</tr>
<tr>
<td>rtl_storesys_get_id_from_desc 415</td>
</tr>
<tr>
<td>rtl_storesys_get_id_from_name 416</td>
</tr>
<tr>
<td>rtl_storesys_get_info_from_desc 417</td>
</tr>
<tr>
<td>rtl_storesys_get_info_from_id 418</td>
</tr>
<tr>
<td>rtl_storesys_get_name_from_desc 419</td>
</tr>
<tr>
<td>rtl_storesys_get_name_from_id 420</td>
</tr>
<tr>
<td>rtl_storesys_get_password_from_desc 421</td>
</tr>
<tr>
<td>rtl_storesys_get_password_from_id 422</td>
</tr>
<tr>
<td>rtl_storesys_get_user_from_desc 423</td>
</tr>
<tr>
<td>rtl_storesys_get_user_from_id 424</td>
</tr>
<tr>
<td>rtl_storesys_is_configured 425</td>
</tr>
<tr>
<td>rtl_storesys_log 426</td>
</tr>
<tr>
<td>rtl_timet_to_rtltime 427</td>
</tr>
<tr>
<td>rtl_tm_to_rtltime 428</td>
</tr>
<tr>
<td>rtl_transport_get_info 429</td>
</tr>
<tr>
<td>rtl_transport_get_name 430</td>
</tr>
<tr>
<td>rtl_transport_is_configured 432</td>
</tr>
<tr>
<td>rtl_transport_log 433</td>
</tr>
<tr>
<td>rtl_transports_get_user_data 431</td>
</tr>
<tr>
<td>rtl_unpack_nullbits 434</td>
</tr>
</tbody>
</table>

B

bad data
  correcting 133
discard table
  creating 133
description 133
  format 133
restrictions 133
  supported data types 134
reloading 133
statistics 133
benefits 1

C

client applications
  C 275
client library 275
connections
  arguments 278
  authorization 279
description 278
  disconnecting 283
  failure 280
  reconnecting 280
disconnecting 283
Java 275
reconnecting 280
writing 275
communications
  high availability communications library 175
TCP/IP 175
configuration files
  creating 22
  description 51
displaying 40
examples 13
reading 29
rtlconfig 13, 22, 51
rtlconfig.db2 13, 22, 51
rtlconfig.db2_simple_schema 22
rtlconfig.dse 13
rtloader.db2 13
samples 22
specifying 29
counters
  arguments 278
  authorization 279
client applications 278
disconnecting 283
enCRYPTed passwords 279
failure 280
reconnecting 280
controlling DSE
  rtldiscarddmp utility 31
rtlmode utility 32
rtloader command 29

D

data feeds
  feed handler interfaces 175
send_ticks program 21
simple, building 21
source, changing 32, 35
data queries 137
data types
  C 83
  field 83
Java 83
record schema 83
SQL 83
DB2 commands
  CREATE SERVER 23
CREATE USER MAPPING 23
CREATE WRAPPER 23
getstatus 115, 119
UPDATE ALTERNATE SERVER 98
DB2 databases
  Data Partitioning Feature (DPF) 129
DB2 databases (continued)
DPF (Data Partitioning Feature) 129
environment variables 18
federation 17
high availability 98
instances 17
partitions
  configuring for an MPP environment 130
  configuring WebSphere MQ 130
Data Partitioning Feature (DPF) 129
distribution keys 129
DPF (Data Partitioning Feature) 129
flushers 129
per-symbol 129
per-tick 129
stores 20
table schema 125
DB2 tables
  creating 20
  schema 125
  symbols 20, 125
ticks 20, 125
debugging 23
developing
  applications 153, 275
  DSE components 153, 175
  feed handlers 153
  transports 163
directory structure 13
disk
  writes, enabling 32, 35
DSE administration 11, 29
DSE applications
developing 153
DSE commands
  rtl_make_pass 31, 49
  rtl discard dmp 31
  rtlmode 31, 32
  rtloader 29
  rtlshm dmp 31, 40
  rtlstat 31, 43
DSE components
developing 153
DSE overview 1
DSE terminology 5

E
ENABLE_NEWSYM configuration parameter 52
environment variables
  setting up 18
error message files
  specifying 29
error messages 441
stderr 29

F
failures
  error recovery 131
resources 119
feed configuration parameters
description 54
FEEDXX_ACCEPT_TICKS_ALWAYS 54
FEEDXX_GAPFILL 54
FEEDXX_HANDLER_NAME 54
feed configuration parameters (continued)
FEEDXX_INFO 55
FEEDXX_SOURCE 55
FEEDXX_STORE_NAME 55
FEEDXX_TARGET 55
FEEDXX_TRANSPORT_NAME 55
FEEDXX_WRITECANBLOCK 56
feed handler configuration parameters
description 56
HANDLERXX_INFO 56
HANDLERXX_LIB_PATH 56
HANDLERXX_MSG_PATH 56
HANDLERXX_NAME 57
feed handlers
  building 161
  control flows
    description 156
    feed handler initialization 157
    feed handler shutdown 158
    feed initialization 157
    feed shutdown 158
    per message processing 157
    processing 156
data handling approaches 154
data processing 155
deployment 161
description 153
  developing 153
error handling 158
error logging 158
examples
  generic 161
  high availability 161
  simple 161
  TAQ 161
framework 153
  high availability, building 106
  message publishing 155
  metadata updates 155
  record structures 154
  reference and context information
    FEEDXX_INFO 156
    HANDLERXX_INFO 156
    user data areas 156
    simple 13
    simple, building 21
tick storage 155
  trace 160
feedback
  sending ix
  feeds
    control flows
      pausing 167
      resuming 167
FEEDXX_ACCEPT_TICKS_ALWAYS configuration parameter 54
FEEDXX_GAPFILL configuration parameter 54
FEEDXX_HANDLER_NAME configuration parameter 54
FEEDXX_INFO configuration parameter 55
FEEDXX_SOURCE 106
FEEDXX_SOURCE configuration parameter 55
FEEDXX_STORE_NAME configuration parameter 55
FEEDXX_TARGET configuration parameter 55
FEEDXX_TRANSPORT_NAME configuration parameter 55
FEEDXX_WRITECANBLOCK configuration parameter 56
FLBS configuration parameter 68

Index 497
functions

G

general system configuration parameters
description 52
ENABLE_NEWSYM 52
INCLUDE 52
NODISKWRITE 52
RESTART_ENABLE 52, 87
RTL_LOG_LOCALTIME 53
RTL_LOG_PATH 53
RTL_SYS_MSG_PATH 53
UDPX 54
group configuration parameters
description 57
GROUPXX_AUTHLEVEL 57
GROUPXX_NAME 57
GROUPXX_PASSWORD 57
group_name
about 279
group_password
about 279
GROUPXX_AUTHLEVEL configuration parameter 57
GROUPXX_NAME configuration parameter 57
GROUPXX_PASSWORD configuration parameter 57
H

HA_COMM_TIMEOUT configuration parameter 58
HA_ENABLE configuration parameter 58
HA_INFO configuration parameter 58
HA_LIB_PATH configuration parameter 58
HA_MAXMSG_SIZE configuration parameter 58
HA_MSG_PATH configuration parameter 59
HA_RETRY_DELAY configuration parameter 59
HA_SAVEBUF_SIZE configuration parameter 59
HA_SHARED_DB configuration parameter 59
HA_STANDBY_API configuration parameter 59
HA_STANDBY_PUB configuration parameter 60
HA_SYNC_DELAY configuration parameter 60
HANDLERXX_INFO configuration parameter 56
HANDLERXX_LIB_PATH configuration parameter 56
HANDLERXX_MSG_PATH configuration parameter 56
HANDLERXX_NAME configuration parameter 57
HASHSIZE configuration parameter 68
HAXX_PEER_INFO configuration parameter 60
HAXX_PEER_NAME configuration parameter 60
help information
displaying 29, 40, 43
high availability
clusters
building, scenario 101
configuration scenario 101
configuring, scenario 103
creating wrapper 118
description 95
failover 95
IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM) 97
IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM), configuring 111
IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM), management script 111
high availability (continued)
clusters (continued)
IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM), preparing 104
installing, scenario 104
management software 97
managing, scenario 101
running queries 118
setup, scenario 106
starting 96
starting, scenario 115
testing, scenario 108, 115
with IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM) 101
DB2 configurations 98
advantages 98
HACMP (High Availability Cluster Multi-Processing) 98
HADR (high availability disaster recovery) 98, 101, 105
High Availability Cluster Multi-Processing (HACMP) 98
high availability disaster recovery (HADR) 98, 101, 105
description 95
instances 96
restartable mode 87
configuration, scenario 89
configuring WebSphere MQ, scenario 91
installing, scenario 90
monitoring tick flow, scenario 93
recovering from failure, scenario 93
running, scenario 89
setting up DB2, scenario 91
setting up DSE, scenario 90
shutting down, scenario 94
with DB2 89
with WebSphere MQ for AIX, Linux, and Solaris 89
scenario
building cluster 101
configuration 101, 103
IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM), configuring 111
IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM), preparing 104
installing 104
managing cluster 101
restartable mode with WebSphere MQ and DB2 89
setting up DSE 106
starting cluster 115
testing 108
testing cluster 115
with IBM Tivoli System Automation for Multiplatforms (IBM Tivoli SAM) 101
states 96
ACTIVE 96
changing 32, 37
displaying 43, 46
INACTIVE 96
specifying 29
STANDBY 96
STARTED 96
SYNCHRONIZING 96
storage systems
description 98
enabling 99
transactional processing 87
high availability configuration parameters
description 57
high availability configuration parameters (continued)

HA_COMM_TIMEOUT 58
HA_ENABLE 58
HA_INFO 58
HA_LIB_PATH 58
HA_MAXMSG_SIZE 58
HA_MSG_PATH 59
HA_RETRY_DELAY 59
HA_SAVEBUF_SIZE 59
HA_MAXMSG_SIZE 58
HA_MSG_PATH 59
HA_RETRY_DELAY 59
HA_SAVEBUF_SIZE 59
HA_SHARED_DB 59, 99
HA_STANDBY_API 59
HA_STANDBY_PUB 60
HA_SYNC_DELAY 60
HAXX_PEER_INFO 60
HAXX_PEER_NAME 60

host_name
about 278

INCLUDE configuration parameter 52
installation 17
C-API 11
DSE server 11
environment variables 18
getting started 17
Java API 11
quick verification 19
tutorial 17
verification 17, 20
wrappers 11
instances
DB2 databases 17
DSE 96
interfaces
feed handlers
description 175
rtl_feed_get_message_fmts 186
rtl_feed_get_publish_message 184
rtl_feed_get_tick_from_message 180
rtl_feed_init 178
rtl_feed_shutdown 179
rtl_feed_update_metadata 182
rtl_handler_init 176
rtl_handler_shutdown 177
generic storage support
description 187
high availability communications
description 175, 243
rtl_ha_init 244
rtl_ha_shutdown 251
rtl_hacomm_connect 245
rtl_hacomm_disconnect 246
rtl_hacomm_listen 247
rtl_hacomm_peer_id 248
rtl_hacomm_receive 249
rtl_hacomm_send 250
partitioning
description 197
rtl_store_tick_allocate_flusher 201
storage systems
description 175, 187
rtl_storesys_close 188
rtl_storesys_init 189
rtl_storesys_open 190
rtl_storesys_shutdown 192
libraries, shared (continued) 175
high availability communications 175
storage system 3
storage systems 175
transport layer 175
locales supported ix
log file
RTL_LOG_PATH configuration parameter 53
log files
messages 439
rtl.log 23

M
MAX_QUERY_CONNECTS configuration parameter 61
MAX_SHMSIZE configuration parameter 68
messages
format 158, 170, 441
high availability communication library 477
pre-built transport library 479
server 441
storage system 472
metadata
help information
displaying 32
queries 282
shared memory
key, specifying 32
shutting down DSE
without flushing 32
updating 217, 282
writes
automatic 32, 34
manual 32, 39
monitoring
rtlstat utility 43
mqsend program 13

N
nicknames, wrapper
requirements 23
NODISKWRITE configuration parameter 52
NUM_INTERVALS configuration parameter 69

O
overview
DSE 1

P
parameters 52, 54, 56, 57, 61, 67, 74, 79, 82
changing dynamically 32, 37
delimited identifiers 85
partitions
configuring for an MPP environment 130
configuring WebSphere MQ 130
flushers 129
per-symbol 129
per-tick 129
passwords
rtl_make_pass utility 49
performance ix
public header files 435
PURGE_HWM configuration parameter 69
PURGE_LWM configuration parameter 69

Q
queries
client applications 280
data range 281
metadata 282
results 282
SQL 137
statistics 283
tick records 281
query server configuration parameters
description 61
MAX_QUERY_CONNECTS 61
QUERY_THREADS 61
RTL_SERVICE_NAME 61
QUERY_THREADS configuration parameter 61

R
record configuration parameters
description 61
RECORDXX_META_SCHEMA 61
RECORDXX_META_SEQUENCE 64
RECORDXX_NAME 61
RECORDXX_SCHEMA 65
RECORDXX_SEQUENCE 67
record types
METADATA 143
RELEASE 143
TICKDATA 143
RECORDXX_META_SCHEMA configuration parameter 61
RECORDXX_META_SEQUENCE configuration parameter 64
RECORDXX_NAME configuration parameter 61
RECORDXX_SCHEMA configuration parameter 65
RECORDXX_SEQUENCE configuration parameter 67
REFRESH_INTERVAL configuration parameter 70
release information
displaying 32, 40, 43
release information, displaying 29
requirements
hardware 10
software 9
system 9
RESTART_ENABLE 87
RESTART_ENABLE configuration parameter 52
restartable mode
collection 89
configuring WebSphere MQ 91
description 87
differences from non-restartable mode 87
enabling 52
installation 90
monitoring tick flow 93
recovering from failure 93
requirements 87
restrictions 87
running 89
scenario 89, 90, 91, 93, 94
setting up DB2 91
setting up DSE 90
shutting down DSE 90
transports
data source switching 166
shared memory configuration parameters
ADD_SHMSIZE 68
description 67
FLBS 68
HASHSIZE 68
MAX_SHMSIZE 68
NUM_INTERVALS 69
PURGE_HWM 69
PURGE_LWM 69
REFRESH_INTERVAL 70
SECHDR_SIZE 70
SHMKEY 70
SHMSIZE 71
SHMKEY configuration parameter 70
SHMSIZE configuration parameter 71
shutting down DSE
after flushing 32, 37
without flushing 40
SQL queries
data flushes 43, 46
discarded data 43
feed handler buffer 43, 45
feed handlers 43, 45
metadata writes 43, 48
partitioner 43
partitioning 47
performance 43, 47
reports 43
shared memory 43, 47
spinlocks 43, 47
stones 43, 45
summary 43, 46, 48
symbols 43, 47
transmit 43, 48
transmit rate 43, 48
zeros 43

storage system configuration parameters
description 71
STORESYSXX_CONN 71, 123
STORESYSXX_HA 71, 99
STORESYSXX_INFO 72, 123
STORESYSXX_LIB_PATH 72, 123
STORESYSXX_MSG_PATH 72, 123
STORESYSXX_NAME 72
STORESYSXX_PASSWORD 73, 123
STORESYSXX_RECONNECT_COUNT 73, 99
STORESYSXX_RECONNECT_PERIOD 73, 99
STORESYSXX_RETRY_COUNT 73, 99
STORESYSXX_RETRY_PERIOD 73, 99
STORESYSXX_USER 73, 123

storage systems
error recovery 131
setup 20
test 17
storagesys_name
about 278
store
bringing offline 32
bringing online 32, 39
taking offline 37

S

scenarios
high availability clusters 101
restartable mode with WebSphere MQ and DB2 87, 89
schema, samples
db2-schema_simple_schema.sql 13
db2-schema.sql 13, 20
sample_wrapper.sql 13, 23
wrapper_schema.sql 13
SECHDR_SIZE configuration parameter 70
send_ticks program 13, 21, 101, 108
service_name
about 278
shared memory
consistency check 40
data structures 40
derived information 40
key, specifying 29, 40, 43, 48
metadata 40
printing 40
rtlshmdmp utility 40
reinitializing 29
ticks 40
ticks, adding 32, 36
store configuration parameters
description 74
STOREXX_AUTHLEVEL 74
STOREXX_DISCARD_ENABLE 74
STOREXX_FLUSH_BATCH 74, 123
STOREXX_FLUSH_PERIOD 74
STOREXX_INFO 75
STOREXX_INSERT_BATCH_SIZE 75, 123
STOREXX_META_BATCH 75, 123
STOREXX_META_MODE 75
STOREXX_META_PERIOD 76
STOREXX_NAME 76, 123
STOREXX_NEWSYMBOLS 76
STOREXX_NODISKWRITE 76
STOREXX_NUM_FLUSHERS 76
STOREXX_RECTYPE 77
STOREXX_SCHEMA 77
STOREXX_SCHEMA_TYPE 77
STOREXX_SYMBOL_BATCH 77
STOREXX_SYMBOL_SET_SIZE 77
STOREXX_SYMBOL_TABLE 78
STOREXX_SYSNAME 78
STOREXX_TICK_SET_SIZE 78
STOREXX_TICK_TABLE 78
STOREXX_TICK_TABLE 78
store_name
about 278
stores
description 20
symbols tables 20
ticks tables 20
STORESYSX_CONN 123
STORESYSXX_CONN configuration parameter 71
STORESYSXX_HA configuration parameter 71
STORESYSXX_INFO 123
STORESYSXX_INFO configuration parameter 72
STORESYSXX_LIB_PATH 123
STORESYSXX_LIB_PATH configuration parameter 72
STORESYSXX_MSG_PATH 123
STORESYSXX_MSG_PATH configuration parameter 72
STORESYSXX_NAME configuration parameter 72
STORESYSXX_PASSWORD 123
STORESYSXX_PASSWORD configuration parameter 73
STORESYSXX_RECONNECT_COUNT configuration parameter 73
STORESYSXX_RECONNECT_PERIOD configuration parameter 73
STORESYSXX_RETRY_COUNT configuration parameter 73
STORESYSXX_RETRY_PERIOD configuration parameter 73
STORESYSXX_USER 123
STORESYSXX_USER configuration parameter 73
STOREXX_AUTHLEVEL configuration parameter 74
STOREXX_DISCARD_ENABLE configuration parameter 74
STOREXX_DISCARD_ENABLE configuration parameter 133
STOREXX_FLUSH_BATCH 123
STOREXX_FLUSH_BATCH configuration parameter 74
STOREXX_FLUSH_PERIOD configuration parameter 74
STOREXX_INFO configuration parameter 75
STOREXX_INSERT_BATCH_SIZE 123
STOREXX_INSERT_BATCH_SIZE configuration parameter 75
STOREXX_META_BATCH 123
STOREXX_META_BATCH configuration parameter 75
STOREXX_META_MODE configuration parameter 75
STOREXX_META_PERIOD configuration parameter 76
STOREXX_NAME 123
STOREXX_NAME configuration parameter 76
STOREXX_NEWSYMBOLS configuration parameter 76
STOREXX_NODISKWRITE configuration parameter 76
STOREXX_NUM_FLUSHERS configuration parameter 76
STOREXX_RECTYPE configuration parameter 77
STOREXX_SCHEMA configuration parameter 77
STOREXX_SCHEMA_TYPE configuration parameter 77
STOREXX_SYMBOL_BATCH configuration parameter 77
STOREXX_SYMBOL_SET_SIZE configuration parameter 77
STOREXX_SYMBOL_TABLE configuration parameter 78
STOREXX_SYSNAME configuration parameter 78
STOREXX_TICK_SET_SIZE configuration parameter 78
STOREXX_TICK_TABLE configuration parameter 78

T

ticks
shared memory 32, 36
trace
buffer contents 40
components 80
description 485
disabling 32, 38, 80, 485
enabling 32, 38, 80, 485
feed handlers 160
file name format 485
levels 80
high 80
low 80
medium 80
off 80
logging 32, 35
macros
F_RETURN(lvl) 160
RETURN_PTR(lvl,v) 160
RETURN(lvl,v) 160
TRACE(lvl,Largs) 160
TRACESUB(lvl,args) 160
TRACEB(lvl,ptr,len) 160
RETURN_PTR(lvl,v) 160
settings
changing 32, 38
displaying 32, 38
transports 170
trace configuration parameters
components 80
description 79
levels 80
TRACEBUFFER_SIZE 79
TRACEFILE_DIR 79
TRACEFILE_SIZE 80
TRACELEVEL_component 80
trace files
specifying 29
trace level configuration parameters 80
trace level configuration parameters
TRACELEVEL_ALL 80
TRACELEVEL_FLUSHER 80
TRACELEVEL_FEED 80
TRACELEVEL_HA 80
TRACELEVEL_PARTITIONER 80
TRACELEVEL_LISTENER 80
TRACELEVEL_MAIN 80
TRACELEVEL_HA_COMM_LIB 80
TRACELEVEL_HANDLER 80
TRACELEVEL_INSERTER 80
TRACEFILE_DIR configuration parameter 79
TRACEFILE_SIZE configuration parameter 80
TRACELEVEL_component configuration parameter
description 80
levels 80
parameters for components 81
transport configuration parameters
description 82
TRANSPORTXX_INFO 82, 172
TRANSPORTXX_LIB_PATH 82
TRANSPORTXX_MSG_PATH 82
TRANSPORTXX_NAME 82
TRANSPORT_MSG_PATH configuration parameter 82
transports
data source switching 166
description 164
pausing feeds 167
per message processing 168
processing 164
resuming feeds 167
target switching 166
transport handler initialization 165
transport handler shutdown 169
transport initialization 165
transport shutdown 169
deployment 171
description 163
developing 153, 163, 171
derror handling 170
derror logging 170
framework 163
pre-built 163
files 163, 171
files, format 171
formats, source and target names 171
messages 171
parameters 172
restrictions 173
TCP/IP sockets 163, 171
TCP/IP sockets, format 171
WebSphere MQ queues 163, 171
WebSphere MQ queues, format 171
properties 164
restartable mode
data source switching 166
per message processing 168
transport shutdown 169
trace 170
TRANSPORTXX_INFO configuration parameter 82, 172
TRANSPORTXX_LIB_PATH configuration parameter 82
TRANSPORTXX_NAME configuration parameter 82
troubleshooting
diagnostic log file 439
resources 439
rtlshmdmp utility 487
utilities
rtl_make_pass 31, 49
rtldiscarddmp 31, 133
rtlmode 31, 32
rtlshmdmp 31, 40
rtlstat 31, 43
Utility programs 31

W
WebSphere MQ
configuring for DSE 91
wrappers 17
creating, scenario 118
debugging 142
description 141
error codes
description 139
installation 11
nicknames 23, 141
description 143
options 143
syntax 143
options 141
query, simple 23
querying, scenario 118
servers 23, 141
description 142
options 142
syntax 142
setup 23
SQL queries 137
syntax 141
trace 141
user mapping 23, 141
description 143
options 143
syntax 143

U
UDF (user-defined functions)
rtlElem 148
UDPX configuration parameter 54
user-defined functions
examples 13
user-defined functions (UDF)
rtlElem 148

Index 503
Program Number: 5799-HGK

Printed in USA