Spectra ORB
C Edition

Lightweight Log Service
Guide

PRISMTECH

Spectra ORB
C Edition

Lightweight Log Service Guide

Part Number: EORBC-LOGLWG

Doc Issue 26, 3 June 2013

Copyright Notice

Contents

Table of Contents

Preface

About the Lightweight Log Service Guide

The *Lightweight Log Service Guide* describes the Spectra ORB Lightweight Log Service C Edition product and how it can be used as a minimal, lightweight logging service for resource-constrained, CORBA-based applications using the Spectra ORB.

The *Lightweight Log Service Guide* is included with the *Spectra ORB Documentation Set* and is intended to be used with the *IDL*, *Reference*, and *User Guides*, as well as the other documents included with the Spectra ORB product. Please refer to the *Product Guide* for a complete list of documents.

Intended Audience

The *Lightweight Log Service Guide* is intended to be used by developers who wish to integrate the Spectra ORB Lightweight Log Service into products which comply with OMG standards for object services. Readers who use this guide should have a good understanding of the relevant programming languages (*e.g.* C, IDL) and of the relevant underlying technologies (*e.g.* CORBA).

Organisation

The Lightweight Log Service Guide is organised into the following sections:

- The *Introduction* provides a general description of the service
- Chapter 1, *Basic Concepts*, describes the basic concepts and architecture
- Chapter 2, *Specific Features*, describes specific features and service's API
- Chapter 3, *Running the Service*, describes how to embed the Spectra ORB Lightweight Log Service into programs and use the Service's standalone executable
- Chapter 4, *Creating Applications*, provides example source code demonstrating how to create an application which uses the Service.

Conventions

The conventions listed below are used to guide and assist the reader in understanding the Lightweight Log Service Guide.


Item of special significance or where caution needs to be taken.


Item contains helpful hint or special information.

WIN
Information applies to Windows (*e.g.* XP, Vista, Windows 7) only.

UNIX
Information applies to Unix-based systems (*e.g.* Solaris) only.

C C language specific.

C++ C++ language specific.

Java Java language specific.

Hypertext links are shown as *blue italic underlined.*

On-Line (PDF) versions of this document: Items shown as cross references to other parts of the document, *e.g. Contacts* on page viii, are hypertext links: jump to that section of the document by clicking on the cross reference.

```
%  Commands or input which the user enters on the
   command line of their computer terminal
```

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown as small `Courier` font in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, “”);
```

Italics and ***Italic Bold*** are used to indicate new terms, or emphasise an item.

Arial Bold is used to indicate user-related actions, *e.g.* **File > Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900
Fax: +44 (0)191 497 9901

Web: *http://www.prismtech.com*
Technical questions: *crc@prismtech.com* (Customer Response Center)
Sales enquiries: *sales@prismtech.com*

Introduction

Description

The Spectra ORB Lightweight Log Service accepts and manages Log records. It is intended to be an efficient, central facility used mainly in high performance, resource constrained environments such as in embedded environments. The Log records which the Lightweight Log Service is designed to store and manage will be produced by applications which reside in the same environment as the Service. The records are stored in a memory-only storage area which is owned and managed by the Service.

The Spectra ORB Lightweight Log Service C Edition is compliant with version 1.1 of the OMG's Lightweight Log Service Specification and satisfies the requirements of the Software Communications Architecture (SCA).

The Spectra ORB Lightweight Log Service can be used in all areas of embedded systems (such as machine control, onboard vehicle systems, pocket computer and electronic organizers) plus in any application area where a small, memory-only logging facility is needed.

OMG Standard Features

The Spectra ORB Lightweight Log Service is designed for use in embedded and real-time systems. The service provides the following OMG specified features:

- Support for simple logging. The Spectra ORB Lightweight Log Service is a lean, stand-alone service targeted for use where resources are constrained, such as in embedded environments.
- Provision of simple *Log producer* and *Log consumer* interfaces for ease of use.
- Logging information is stored only in memory in order to accommodate constrained environments: no persistent storage is supported nor required.
- The data structure which is used to store a Log record is specially designed to accommodate the constraints of embedded environments. This structure, combined with a list of well known typecodes, provides the control on type variety which is needed in an embedded system. Further, the structure does not use *any*s, thereby simplifying use with embedded ORBs (which frequently impose restrictions on the any type).
- Log records can be read as a series of small, consecutive groups for efficiency, similar to using an iterator.
- Log write operations are strictly asynchronous. The Log does not transmit feedback or exceptions: this avoids any interference to Log producer timing constraints.

Log Service

CHAPTER

1 Basic Concepts

This section describes the basic concepts and architecture of the Lightweight Log Service as defined in the OMG's Lightweight Log Service Specification.

1.1 Architecture

1.1.1 General Organisation

The Lightweight Log Service is a self-contained service. It does not use or rely on event channels or other infrastructure, unlike the OMG's *Telecom Log Service* which is layered on top of the *Notification Service*. The Lightweight Log Service is specifically designed for resource constrained environments.

A Lightweight Log Service instance (Log) is an implementation detail. There is no single interface in the IDL which represents a Log. Instead, a Log instance realises three distinct interfaces, each implemented as separate CORBA objects, corresponding to a different client role:

- `LogProducer` - This interface is for use by clients which produce Log records. It declares operations for writing Log records into the associated Log.
- `LogConsumer` - This interface is for use by clients which consume or query the Log records which are stored in the associated Log. It declares operations for querying and retrieving Log records according to various criteria.
- `LogAdministrator` - This interface is for use by clients which manage the operational state of a Log. It declares operations for setting the various administrative options, and for clearing or destroying the associated Log.

This approach to interface design is a departure from the majority of OMG service specifications with which the reader may be familiar. The typical approach, taken in other OMG service specifications is to declare separate aspects in separate interfaces, but then to inherit them all into one large interface. This enables clients to access all the supported interfaces of an object through a single object reference.

All three interfaces inherit the `LogStatus` interface. The `LogStatus` interface provides access to status information for the associated Log. This enables all clients to have easy access to status information for the Log.

As of version 1.1 of the specification a new interface `Log` has been defined that inherits from the main three functional interfaces of the service (producer, consumer and administrator). The allows a single object reference to be used to resolve the service, from which the individual log component interfaces can be found.

1.1.2 Log Records

A client which writes to a `Log`, does so by using the `ProducerLogRecord` struct to pass the Log record information. This struct declares the following fields:

- *`producerId`* - a textual identifier for the producer
- *`producerName`* - a textual name for the producer
- *`level`* - Log record classification according to the `LogLevel` type
- *`logData`* - the informational message to be logged.

When a `Log` receives a `ProducerLogRecord` from a producer, it wraps it in a `LogRecord` structure, fills in some additional information, and then stores it. The `LogRecord` structure declares the following fields:

- *`id`* - uniquely identifies the Log record within the context of the Log
- *`time`* - the time stamp for the record
- *`info`* - the `ProducerLogRecord` (from the producer)

Figure 1, *Lightweight Log Service*, shows the relationship between these components, along with the other Lightweight Log Service components.

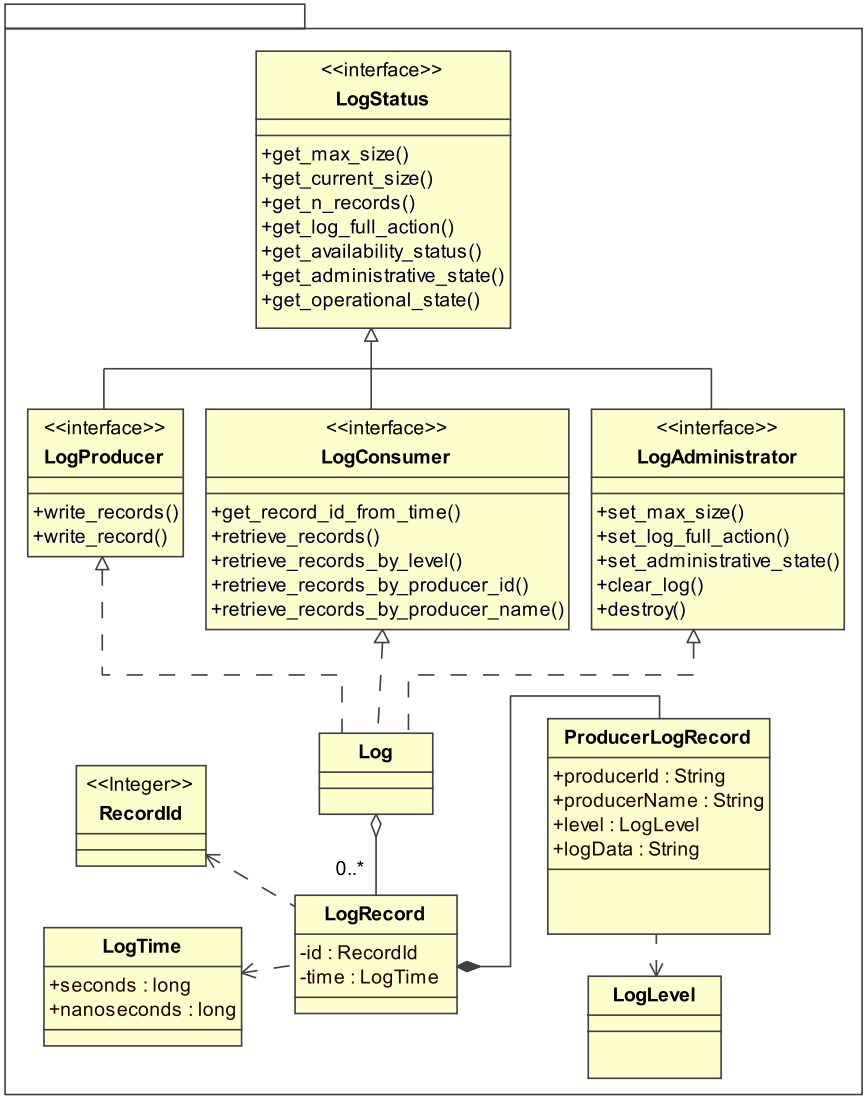

Figure 1 Lightweight Log Service

1.1.3 Log Lifecycle

Log creation and destruction is straightforward:

1. A Log instance is created by an application using the LwLogService::init() method of the service's Embedding API (see Section 3.1, *Embedding the Service*, on page 25). This operation returns three distinct CORBA object references, each one supporting one of the three interfaces onto the Log (LogProducer, LogConsumer and LogAdministrator).
2. During the life of a Log, its clients interact with it via one or more of its three interfaces.
3. A Log can be destroyed using the destroy() operation on its LogAdministrator interface. This operation destroys the Log, its associated storage area, and any Log records that it may contain. It also destroys the three objects that implement the Log interfaces.

1.1.4 LogRecord Lifecycle

1. A client which produces Log records does so by creating a `ProducerLogRecord`, containing the information to be logged. It then invokes one of the write operations on the `LogProducer` interface.
2. When a Log receives a `ProducerLogRecord` (via its `LogProducer` interface), it wraps it in a `LogRecord` structure and fills in the id and time members.
3. The Log then stores that `LogRecord` in its memory-based Log storage area if the Log is able or allowed to accept it. Whether the Log record can be stored or not depends on the administrative and operational settings of the Log.
4. A consumer retrieves `LogRecord`s, from a Log, using the operations on the `LogConsumer` interface. The `LogConsumer` interface declares a number of operations for retrieving Log records according to various search criteria.
5. Records can be removed from a Log, using the `clear_log()` operation on the `LogAdministrator` interface. This operation removes all of the records from the Log, reducing the current size of the Log to zero, but leaving the maximum size unaffected.
6. Although there are no operations available to selectively remove Log records from a Log, a Log’s administrative properties can be set to allow new records to overwrite the oldest records in order to reuse the existing storage space.

CHAPTER

2 Specific Features

This section describes the Spectra ORB Lightweight Log Service's specific features and API.

This section covers the following Log components, in order:

- *Data types and structures, including constants, typedefs, enumerations and structs. Items covered include Log levels, administrative and operational states' values, Log time and Log record structures.*
- *Interfaces and their associated operations, including:*
  - `LogStatus` *(the base-interface for the remaining interfaces)*
  - `LogProducer`
  - `LogConsumer`
  - `LogAdministrator`
- *Exceptions*

The components described here include:

- *those which comply with the OMG Lightweight Log Service Specification*
- *PrismTech-specific components which are in addition to those specified in the OMG's specification.*

The OMG Lightweight Log components are in the `CosLwLog` *module.*

The PrismTech Lightweight Log components are in the `EORB_LwLogService` *module.*

The components which are PrismTech-specific are noted as such in the text.

The API definitions generally follow the convention for the C language, although the namespace component of the interface and other names may be omitted for simplicity and readability. For example, the following C interface definitions

CosLwLog_LogStatus_get_operational_state ()
CosLwLog_OperationalState CosLwLog_LogStatus_get_operational_state ()

are shown as simply

get_operational_state ()
OperationalState get_operational_state ()

2.1 Data Types and Structures

2.1.1 Constants

LogLevel Constants

The following constants are used to classify Log records. The value is saved as a `LogLevel` type. See *typedef unsigned short LogLevel* on page 12.

SECURITY_ALARM	= 1
FAILURE_ALARM	= 2
DEGRADED_ALARM	= 3
EXCEPTION_ERROR	= 4
FLOW_CONTROL_ERROR	= 5
RANGE_ERROR	= 6
USAGE_ERROR	= 7
ADMINISTRATIVE_EVENT	= 8
STATISTIC_REPORT	= 9

Codes 10-26 are reserved for program debugging levels.

2.1.2 TypeDefs

typedef unsigned short LogLevel

The `LogLevel` type is used to classify Log records. A `LogLevel` value is saved in the Log record and is available to the consumer clients when the record is retrieved. The value has no particular meaning nor has any side effects during storage of the record in the Log. See *LogLevel Constants* on page 12.

typedef sequence <CosLwLog_LogLevel> LogLevelSequence

Defines an unbound sequence of `LogLevel` values.

typedef sequence <CosLwLog_LogRecord> LogRecordSequence

Defines an unbounded sequence of `LogRecords`.

typedef sequence <CosLwLog_ProducerLogRecord> ProducerLogRecordSequence

Defines an unbound sequence of `ProducerLogRecords`.

typedef unsigned long long RecordId

A 64-bit integer which is used to hold the unique ID value for a Log record.

typedef sequence <string> StringSeq

A sequence of *producerId*s. See *struct ProducerLogRecord* on page 15.

2.1.3 Enumerations

enum AdministrativeState

```
enum AdministrativeState {
   locked,
   unlocked
}
```

The AdministrativeState enumerationsLog indicate whether the LogLog will accept and store Log records from Log record producers or not.

unlocked - the Log will accept Log records for storage

locked - the Log will not accept new Log records for storage. Records which are already in the Log can still be read or deleted

enum LogFullAction

```
enum LogFullAction {
   WRAP,
   HALT
}
```

The LogFullAction values are used by *set_log_full_action()* to determine which action to take when the Log becomes full. Also see *set_log_full_action ()* on page 23.

HALT - no more Log records are allowed to be placed into the Log

WRAP - Log records can continue to be placed into the Log; new Log records will overwrite the space occupied by the oldest Log records

enum OperationalState

```
enum OperationalState {
   disabled,
   enabled
}
```

The OperationalState enumeration defines the Lightweight Log Service's operational states.

enabled - signifies that the Log is available for use by Log record producer and consumer clients

disabled - signifies that the Log has encountered a run time problem and is not available for use by Log record producers or consumers.

2.1.4 Structs

struct AvailabilityStatus

```
struct AvailabilityStatus {
   boolean off_duty;
   boolean log_full;
};
```

The `AvailabilityStatus`' members indicate if the Log is available for use.

`off_duty` - indicates that the Log's `AdministrativeState` is *`locked`* or the `OperationalState` is *`disabled`*, in other words, the service is not available for use, when this is set to `TRUE`.

`log_full` - indicates the Log is full when this is set to `TRUE`.

struct EORB_LwLogService_Config

```
struct EORB_LwLogService_Config {
   unsigned long boolean qosMaxSize;
   boolean qosWithLocking;
};
```

i This is a *PrismTech*-specific structure which is used when creating a Log. This struct holds values which determine certain aspects of the Log's behaviour.

`qosMaxSize` - Specifies the maximum size, in bytes, that will be used by the Log storage area, for storing Log records. The default value is 2048 bytes

`qosWithLocking` - Controls whether the implementation uses locking (`TRUE`) or not (`FALSE`). If locking is turned off, then concurrent invocations on the service may result in undefined behaviour. The default value of this property is `TRUE`(1).

struct LogRecord

```
struct LogRecord {
   CosLwLog_RecordId id;
   CosLwLog_LogTime time;
   CosLwLog_ProducerLogRecord info;
};
```

`LogRecord` is the data type which is placed into the Log. *`LogRecord`* contains the Log information created by a Log producer, plus additional information which identifies the Log record and when it was placed into the Log.

`id` - a value which uniquely identifies the Log record

`time` - the time that the Log record has been created and subsequently placed into the Log

info - the *ProducerLogRecord* created by a Log producer. See *struct ProducerLogRecord* below.

struct LogTime

```
struct LogTime {
   long seconds;
   long nanoseconds;
};
```

The time format used by `LogRecord` to record when a Log record was created. The `LogTime` fields map directly to the POSIX *timespec* structure.

Note that on some embedded systems it may not be possible to obtain the actual system time, in which case the time field for a log entry may be set to have values of zero or possibly some other value, such as clock ticks.

struct ProducerLogRecord

```
struct ProducerLogRecord {
   string producerId;
   string producerName;
   CosLwLog_LogLevel level;
   string logData;
};
```

ProducerLogRecord is the data type used by Log producer clients to contain the producer's Log data. The `ProducerLogRecord` is encapsulated in a *LogRecord* struct before it is stored in the Log.

producerId - a string value which identifies the Log producer

producerName - the Log producer's name

level - a classification value indicating the type `LogRecord` the Log is, such as if it is a security alarm Log, failure alarm Log, etc. See *LogLevel Constants* on page 12.

logData - textual information which the Log producer stores in the Log record

struct EORB_LwLogService_References

```
struct EORB_LwLogService_References {
   CosLwLog_LogConsumer consumer;
   CosLwLog_LogProducer producer;
   CosLwLog_LogAdministrator administrator;
};
```

i This is a *PrismTech*-specific structure which is returned by the *EORB_LwLogService_init()* when creating a Log.

The *References'* members contain object references to the Log's interfaces.

consumer - reference to the LogConsumer object

producer - reference to the LogProducer object

administrator - reference to the LogAdministrator object

2.2 Interfaces

interface LogStatus

The *LogStatus* interface provides common operations which are inherited by the other Lightweight Log Service interfaces.

get_administrative_state ()

```
CosLwLog_AdministrativeState get_administrative_state ()
```

This operation gets the value indicating whether the Log is allowed to accept new Log records or not.

Returns an AdministrativeState value (see *enum AdministrativeState* on page 13) for the description of AdministrativeState values.

get_availability_status ()

```
CosLwLog_AvailabilityStatus get_availability_status ()
```

This operation obtains the availability status of the Log. The availability status indicates whether the Log is able to able to accept Log records or not.

Returns an AvailabilityStatus value (see *struct AvailabilityStatus* on page 14) for the description of AvailabilityStatus values.

get_current_size ()

```
unsigned long long get_current_size ()
```

Log records are stored in a storage area encapsulated by the Log class. This operation returns the size, in bytes, of the Log currently occupied by Log records. This value is less than or equal to the total storage area size returned by the get_max_size() operation.

get_log_full_action ()

```
CosLwLog_LogFullAction get_log_full_action ()
```

get_log_full_action() gets the type of action which is to be taken when the Log is full.

Returns a LogFullAction value (see *enum LogFullAction* on page 13) for the description of LogFullAction values.

get_max_size ()

```
unsigned long long get_max_size ()
```

Log records are stored in the Log's storage area. This operations returns the maximum size, in bytes, that the storage area is allowed to be. See *set_max_size ()* on page 23.

get_n_records ()

```
unsigned long long get_n_records ()
```

The `get_n_records()` operation returns the number of Log records currently stored in the Log.

get_operational_state ()

```
CosLwLog_OperationalState get_operational_state ()
```

This operation gets a value which indicates whether the Log can be accessed or not, for both Log producers and consumers.

Returns an `OperationalState` value (see *enum OperationalState* on page 13) for the description of `OperationalState` values.

interface LogProducer

This interface is used by Log producers to save or write Log records to a Log.

There is no guarantee that a Log record will be accepted or stored by the Log.

write_record ()

```
oneway void write_record (
   in CosLwLog_ProducerLogRecord record)
```

This operation stores or writes a Log record to the Log. A Log record will only be stored when

- the *`AdministrativeState`* is set to *`unlocked`*
- the *`OperationalState`* is set to *`enabled`*
- the *`AvailabilityStatus`*' *`off_duty`* value is `FALSE`

and

- the size of the Log record is smaller than the amount of free space in the Log *or*
- the size of the Log record is greater than the amount of free space in the Log storage area *and* the value of *`LogFullAction`* is set to *`WRAP`*.

If there is not enough space in the store to save the Log record and `LogFullAction` is set to *`HALT`*, then `write_record()` will set the `AvailabilityStatus`' `log_full` value to `TRUE`.

`writeRecord()` places the record passed to it (*`ProducerLogRecord`*) into a *`LogRecord`* struct: the *`LogRecord.time`* field is set to the current UTC time and the *`LogRecord.id`* field is set to a value which uniquely identifies the Log.

Parameters

`record` - the Log record to be stored

write_records ()

```
oneway void write_records (
   in CosLwLog_ProducerLogRecordSequence records)
```

The *`write_records()`* operation writes a sequence of Log records to the Log. `write_records()` behaviour and is the same `write_record()`, notwithstanding that `write_records()` saves a *sequence* of records. (Refer to *write_record ()* above).

Log records (in the sequence passed to `write_records()`) will be saved to the Log until the available storage space becomes less than the size of current record which `write_records()` is attempting to save. For example, if `write_records()` is passed a sequence of ten records and the record store runs out of space while trying to save the seventh record, then the seventh and subsequent records will not be saved, whereas the first six records will have been saved.

Parameters

`records` - sequence of records to be saved or written to the Log

interface LogConsumer

The *`LogConsumer`* interface enables Log records to be retrieved from the Log by Log consumers.

get_record_id_from_time ()

```
CosLwLog_RecordId get_record_id_from_time (
   in CosLwLog_LogTime fromTime)
   raises(CosLwLog_InvalidParam)
```

The `get_record_id_from_time()` operation returns the record ID of the first record in the Log which has a time stamp that is greater than or equal to the time specified in the `fromTime` parameter.

If the Log does not contain a record that meets this criteria, then a record ID will be returned for the next logical record which would have been placed in the store, noting that this *future* record will not have been placed in the store yet. If a retrieval operation attempts to retrieve a record using the ID for this record and it has not yet been placed in the Log then an empty record will be returned.

If the time specified in the `fromTime` parameter is in the future, then there is no guarantee that the records returned by a retrieval operation will have a time stamp that satisfy the time criteria, in other words the returned records could be empty.

Parameters

`fromTime` - specifies which records to retrieve where the record's time stamp must be greater than or equal to the time specified

retrieve_records ()

```
CosLwLog_LogRecordSequence retrieve_records (
   inout CosLwLog_RecordId currentId,
   inout unsigned long howMany)
   raises(CosLwLog_InvalidParam)
```

This operation retrieves a sequence of Log records from the Log. The first record to be retrieved is specified by the *`currentId`* parameter; the number of records to be retrieved is specified by *`howMany`* parameter. If the number of records specified by `howMany` is greater than number of records available, then only the available records will be returned.

The `currentId` and `howMany` values are updated when each record is retrieved:

- `currentId` is set to the record ID of the next record, unless there are no further records, in which case `currentId` is set to zero (0)
- `howMany` is set to the total number of records which have been retrieved

If the record specified by `currentId` does not exist, but corresponds to the next record that will be recorded in the future, `retrieveRecords()` returns an empty sequence of `LogRecords`, sets `howMany` to zero and leaves the value of `currentId` unchanged.

If the record specified by `currentId` does not exist and does not correspond to the next record that will be recorded in the future, or if the Log is empty, then `retrieveRecords()` returns an empty sequence of `LogRecords` and sets `currentId` and `howMany` to zero (0).

Parameters

`currentId` - the first record to be retrieved from the Log

`howMany` - the number of records to be retrieved from the Log

retrieve_records_by_level ()

```
CosLwLog_LogRecordSequence retrieve_records_by_level (
   inout CosLwLog_RecordId currentId,
   inout unsigned long howMany,
   in CosLwLog_LogLevelSequence valueList)
   raises(CosLwLog_InvalidParam)
```

This operation retrieves, from the Log, the number of records specified by *howMay* parameter which have a Log level value appearing in the list of Log levels specified by *valueList*. The first Log record which will be retrieved is specified by the *currentId*.

If the number of records specified by `howMany` is greater than number of records available, then only the available records will be returned.

The `currentId` and `howMany` values are updated when each record is retrieved:

- `currentId` is set to the record ID of the next record
- `howMany` is set to the total number of records which have been retrieved

If no further records are available, then `currentId` is set to the next record that will be recorded in the future the `retrieve_records_by_level()` returns an empty sequence of `LogRecords`, sets `howMany` to zero and leaves the value of `currentId` unchanged.

If the record specified by `currentId` does not exist and does not correspond to the next record that will be recorded in the future, or if the Log is empty, then `retrieve_records_by_level()` returns an empty list of `LogRecords` and sets `currentId` and `howMany` to zero (0).

`retrieve_records_by_level()` does not guarantee to return a sequence of Log records nor to update the `currentId` value. Consequently, the record ID of the first record to be retrieved should be re-established before subsequent invocations of this operation when using a different `valueList` or when using other retrieval operations.

Parameters

currentId - record ID of the starting record

howMany - specifies the number of records to retrieve; during record retrieval its value is set to the number of records actually retrieved

valueList - the Log levels to be searched

retrieve_records_by_producer_id ()

```
CosLwLog_LogRecordSequence retrieve_records_by_producer_id (
   inout CosLwLog_RecordId currentId,
   inout unsigned long howMany,
   in CosLwLog_StringSeq valueList)
   raises(CosLwLog_InvalidParam)
```

This operation retrieves a sequence of records from the Log which have a producer ID that is listed in the *valueList* parameter. The *currentId* identifies first record to be retrieved; *howMany* specifies the number of records to be retrieved. If the number of records specified by howMany is greater than number of records available, then only the available records will be returned.

The currentId and howMany values are updated when each record is retrieved:

- currentId is set to the record ID of the next record
- howMany is set to the total number of records which have been retrieved

If no further records are available, then currentId is set to the next record that will be saved in the future, the operation returns an empty list of LogRecords, sets howMany to zero and leaves the value of currentId unchanged.

If the record specified by currentId does not exist and does not correspond to the next record that will be recorded in the future, or if the Log is empty, then retrieve_records_by_producer_id() returns an empty list of LogRecords and sets currentId and howMany to zero (0).

retrieve_records_by_producer_id() does not guarantee to return a sequence of Log records nor to correctly update the currentId value. Consequently, the record ID of the first record to be retrieved should be re-established before subsequent invocations of this operation when using a different valueList or when using other retrieval operations.

Parameters

currentId - record ID of the starting record

howMany - specifies the number of records to retrieve; during record retrieval its value is set to the number of records actually retrieved

valueList - the producer IDs to be searched

retrieve_records_by_producer_name ()

```
CosLwLog_LogRecordSequence retrieve_records_by_producer_name
(
   inout CosLwLog_RecordId currentId,
```

```
inout unsigned long howMany,
in CosLwLog_StringSeq valueList)
raises(CosLwLog_InvalidParam)
```

This operation retrieves a sequence of records from the Log which have a producer name that is listed in the *valueList* parameter. The *currentId* identifies first record to be retrieved; *howMany* specifies the number of records to be retrieved. If the number of records specified by howMany is greater than number of records available, then only the available records will be returned.

The currentId and howMany values are updated when each record is retrieved:

- currentId is set to the record ID of the next record
- howMany is set to the total number of records which have been retrieved

If no further records are available, then currentId is set to the next record that will be saved in the future, the operation returns an empty list of LogRecords, sets howMany to zero and leaves the value of currentId unchanged.

If the record specified by currentId does not exist and does not correspond to the next record that will be recorded in the future, or if the Log is empty, then retrieve_records_by_producer_name() returns an empty list of LogRecords and sets currentId and howMany to zero (0).

retrieve_records_by_producer_name() does not guarantee to return a sequence of Log records nor to update the currentId value. Consequently, the record ID of the first record to be retrieved should be re-established before subsequent invocations of this operation when using a different valueList or when using other retrieval operations.

Parameters

currentId - record ID of the starting record

howMany - specifies the number of records to retrieve; during record retrieval its value is set to the number of records actually retrieved

valueList - the producer names to be searched

interface LogAdministrator

The *LogAdministrator* interface provides the management functionality which is needed to operate and manage a Log.

clear_log ()

```
void clear_log ()
```

This operation removes all Log records from the Log. The maximum size of the storage area is not changed.

destroy ()

```
void destroy ()
```

This operation destroys the Log and removes it from memory, releasing associated memory resources.

Use `destroy()` with care since any Log records which are in the Log when `destroy()` is called will be lost.

set_administrative_state ()

```
void set_administrative_state (
   in CosLwLog_AdministrativeState state)
```

Sets the administrative state of the Log. The administrative state determines whether or not Log records can be saved to the Log. The required administrative state, passed to the operation via *`state`* parameter and can be:

- *`UNLOCKED`* - Log records are allowed to be placed into the Log and Log records can be read from the Log
- *`LOCKED`* - Log records are not allowed to be placed into the Log; Log records can still be read from the Log

Also see *enum AdministrativeState* on page 13.

Parameters

`state` - the required administrative state

set_log_full_action ()

```
void set_log_full_action (
   in CosLwLog_LogFullAction action)
```

Sets the action to be taken when the Log becomes full. The action to be taken is passed via the *`action`* parameter. The available actions can be:

- *`HALT`* - no more Log records are allowed to be placed into the Log
- *`WRAP`* - Log records can continue to be placed into the Log; new Log records will over write the space occupied by the oldest Log records

Also see *enum LogFullAction* on page 13.

Parameters

`action` - The action to be taken when the Log becomes full.

set_max_size ()

```
void set_max_size (
   in unsigned long long size)
   raises(CosLwLog_InvalidParam)
```

This operation sets the maximum size, in `bytes`, of the Log. It raises the *`InvalidParam`* exception if the supplied parameter (*`size`*) is invalid.

Care should be taken when using `set_max_size()` to avoid allocating a maximum size storage which impinges on or exceeds the amount of memory available on the platform. Further, the maximum size might be highly contrained on certain memory-restricted platforms, therefore particular care is needed in setting the maximum memory sized in these cases.

Parameters

`size` - the size, in bytes, to set the maximum Log size to.

2.3 Exceptions

The `CosLwLog` module defines a single exception, *`InvalidParam`*, described below.

exception InvalidParam

This exception is thrown when a operation is passed an invalid parameter.

CHAPTER

3 Running the Service

The Lightweight Log Service is usually run by programatically embedding it into an executable or module, using the LwLogService embedding API. For convenience, a standalone executable is also provided so that it can be run from the command line.

This section describes how to embed the Lightweight Log Service using its embedding API, plus it describes how to run the standalone executable from the command line.

3.1 Embedding the Service

Refer to the *Install Guide* for specific compiler and linking details for your platform.

The Spectra ORB Lightweight Log Service, as described in previous sections, consists of three interfaces (`LogProducer`, `LogConsumer` and `LogAdministrator`), a Log record storage area and other components which are used to configure and control the Service's behaviour.

The tasks described below show how to embed a Spectra ORB Lightweight Log Service into a code module by setting the configuration component and retrieving references to the interfaces that store, retrieve and administer Log records.

Step 1: Make the Log's embedding API available by placing the following *include* statements in the source code:

```
#include "CosLwLog.h"
#include "eOrbC/EORB/LwLogService.h"
```

i Ensure your build system has `$(EORBHOME)/include/services/lw` on its include path.

Step 2: The configurable aspects of the Log's behaviour are determined by the values of the members of an `EORB_LwLogService_Config` `struct`.
Create a `EORB_LwLogService_Config` variable and set its members to the appropriate values shown under *struct EORB_LwLogService_Config* on page 14.

Step 3: Initialise the Log by calling the `EORB_LwLogService_init()` method. The `EORB_LwLogService_init()` method returns a `struct`, of type `EORB_LwLogService_References`. The `References` `struct` contains

references to the Log's three interfaces, `LogProducer`, `LogConsumer` and `LogAdministrator`, which respectively are used to create and retrieve the Log's records and to administer the Log.

Step 4: Call the appropriate `LogProducer`, `LogConsumer` and `LogAdministrator`, operations to perform required logging operations.

3.1.1 Configuration Properties

The *`EORB_LwLogService_Config`* `struct` declares the `qosMaxSize` and `qosWithLocking` configuration properties to control aspects of the Log's behaviour: `qosMaxSize` Specifies the maximum size, in bytes, that can be used to store Log records; `qosWithLocking` controls whether locking is used or not. For a complete description of these properties see *struct EORB_LwLogService_Config* on page 14.

An example of how to set these configuration properties is shown in *Example 1*, below.

Example 1 Setting Configuration Properties

```
EORB_LwLogService_Config config;

config.qosMaxSize = 2048;
config.qosEntryLocking = TRUE;
```

3.1.2 The init() Method

The Spectra ORB Lightweight Log Service's initialisation method, *`EORB_LwLogService_init()`* initialises a Log instance. The `init()` method takes two parameters:

- a reference to the `CORBA_ORB` that the Log will run in
- a reference to the `PortableServer_POA` that the service will run in
- a *`EORB_LwLogService_Config`* struct containing configuration properties (see *struct EORB_LwLogService_Config* on page 14).

The `init()` method returns a struct containing references to the Log interfaces, *`LogProducer`*, *`LogConsumer`* and *`LogAdministrator`*. See *struct EORB_LwLogService_References* on page 15 for details.

Example 2 Using the init() Method

The following code extract creates a Lightweight Log instance using the *`EORB_LwLogService_init()`* method. The `init()` method is passed a reference to an initialised ORB object, either an initialized POA or `NULL` and an `EORB_LwLogService_Config` `struct` (see *struct EORB_LwLogService_Config* on page 14). If the POA is `NULL`, then the service will resolve the `RootPOA` from the ORB.

```
EORB_LwLogService_References * refs = EORB_LwLogService_init (
   orb, poa, &config, &env);
```

3.2 Running from the Command Line

The Service can run from the command line using the *lwlogc* example executable with zero or more of the options listed in *Table 1*.

```
%  lwlogc [options]
```

The complete source code for `lwlogc` is shown after *Table 1*.

Table 1 Command Line Options

Option	Description
-LogServiceMaxSize <num>	Sets the maximum size, in bytes, of the log record storage area. The default value is *1024*. This option corresponds to the qosMaxSize configuration property in the `EORB::LwLogService::Config` struct.
-LogServiceEntryLocking <on | off>	Turns locking on or off in the service implementation. The default value is *on*. This option corresponds to the qosWithLocking configuration property in the `EORB::LwLogService::Config` struct.
`-LogServiceUIOP`	Runs the service on a UIOP endpoint. Only available on systems where UIOP is a supported transport. The default is `no`.

3.2.1 Running on a Fixed Endpoint

The server can be run on a fixed endpoint by running with the `-ORBPOAEndpoints` argument. The Log Service servant is created within a child POA *LogService* so for example can be run on a fixed IIOP endpoint with:

```
-ORBPOAEndpoints LogService:iiop:<host>:<port>
```

and resolved as an initial reference by a client using:

```
-ORBInitRef LogService=corbaloc:iiop:<host>:<port>/LogService
```

3.2.2 lwlogc Source Code

The source code for creating a log service executable can be found in the provided code examples (`examples/c/lwlog/server/server.c`).

CHAPTER

4 Creating Applications

The main tasks which may normally be performed when using the Spectra ORB Lightweight Log Service include:

- *creation of a Log service server for storing records of data or events sent by clients*
- *creation of clients which supply the data or events*
- *creation of clients which use the Log's records*

This section, Creating Applications, describes how the specific features and requirements for the Spectra ORB Lightweight Log Service can be used to achieve the tasks listed above. The section is organised into a sequence of topics which describes how to

- *create a Log service server*
- *create a combined client which demonstrates how to write and read data to and from the Log server*[1]

The topics use examples to illustrate how relevant tasks can be achieved.

4.1 Server

The following code demonstrates how to create a Lightweight Log Service server.

4.1.1 The IDL File

An IDL file, `lwlog.idl` in this example, defines the Log service example's interface. The interface contains a single operation, `shutdown()`, which is used to shutdown or stop the server.

```
interface lwlog
{
   oneway void shutdown();
};
```

The IDL generated server files, `lwlogS.h` and `lwlogS.c`, used in the example are produced when this file is compiled using the `idlc` complier. The `lwlogS.h` file will be imported into the developer-written server implementation files, `lwlog_i.h` and `lwlog_i.c`, shown below.

1. It should be appreciated that in a real, production environment writing and reading Log data would more likely be performed by separate clients.

4.1.2 Developer-written Implementation Files

4.1.2.1 The Implementation File

The server implementation file, *server.c*, configures, instantiates, runs and shuts down the Log server.

The Includes

The implementation file includes the following header files:

- *LwLogService.h* - for the Lightweight Log Service
- *lwlogS.h* - the idlc generated *implementation base header file*. This file provides C types and the *EPV* and *VEPV* structs that you use in your object implementation. An object implementation file includes this header file, which in turn includes the interface header file, *lwlog.h*.

```
#include "eOrbC/EORB/LwLogService.h"
#include "lwlogS.h"
```

Example Application Servant

The orb declaration and `lwlog_shutdown_i()` definition:

```
static CORBA_ORB orb;

void lwlog_shutdown_i
(
   PortableServer_Servant servant,
   CORBA_Environment * ev
)
{
#if EORB_USE_STDIO
   fprintf (stdout, "Server exiting\n");
   fflush (stdout);
#endif
   CORBA_ORB_destroy (orb, ev);
}
```

Main

The main operation is defined using the Spectra ORB *EORB_MAIN* macro, which is for code portability for systems without a main entry point.

The following essential variables are declared in the main function, along with connecting the TCP, IIOP and POA plugins:

- *config* - a *PrismTech*-specific structure holding values which determine certain aspects of the Log's behaviour
- *ref* - an object reference to the Log interface
- *poa* - a pointer to the ORB's Portable Object Adapter instance

- *oid* - a pointer to the object id of the Log servant associated with the POA instance, poa

```
EORB_MAIN (server)
{
   EORB_LwLogService_Config config;
   PortableServer_ObjectId * oid;
   PortableServer_POA poa;
   CORBA_Environment env;
   CosLwLog_Log * ref;
   CORBA_Object obj;
   POA_lwlog * servant;

   config.qosEntryLocking = TRUE;
   config.qosMaxSize      = 1024;

   /* Install plugins */

   EORB_IIOP_plugin ();
   EORB_POA_plugin ();
   EORB_Stdio_plugin ();
   EORB_File_plugin ();

#if EORB_USE_STDIO
   fprintf (stdout, "LwLog example starting\n");
   fflush (stdout);
#endif
```

Initialisation

The ORB, servant and Log service are initialised.

```
   /* Initialize the ORB */

   orb = CORBA_ORB_init (&argc, argv, "eorb-ce", &env);
   EORB_CHECK_EXC_RETURN_VAL ("ORB_init", &env, -1);

   /* Get the root POA */

  poa = CORBA_ORB_resolve_initial_references (orb, "RootPOA", &env);
  EORB_CHECK_EXC_RETURN_VAL ("resolve_initial_references", &env, -1)

   /* Initialize the servant */

   servant = POA_lwlog__alloc ();
   POA_lwlog__init (servant, &env);
   EORB_CHECK_EXC_RETURN_VAL ("POA_lwlog__init", &env, -1);

   /* Activate the servant */

   oid = PortableServer_POA_activate_object (poa, servant, &env);
   EORB_CHECK_EXC_RETURN_VAL ("POA_activate_object", &env, -1);
   CORBA_free (oid);

   /* Get reference for servant */

   obj = PortableServer_POA_servant_to_reference (poa, servant,
&env);
   EORB_CHECK_EXC_RETURN_VAL ("POA_servant_to_reference", &env, -1);
```

```
   /* Register the reference */

   CORBA_ORB_register_initial_reference (orb, "server", obj, &env);
   EORB_CHECK_EXC_RETURN_VAL ("register_initial_reference", &env,
-1);

   CORBA_Object_release (obj, &env);

   /* Initialize lwlog service */

   ref = EORB_LwLogService_init (orb, NULL, &config, &env);
   EORB_CHECK_EXC_RETURN_VAL ("EORB_LwLogService_init", &env, -1);

   /* Register the references */

   CORBA_ORB_register_initial_reference (orb, "LogService", ref,
&env);
   EORB_CHECK_EXC_RETURN_VAL ("register_initial_reference", &env,
-1);

   CORBA_Object_release (ref, &env);

#if EORB_USE_STDIO
   fprintf (stdout, "Waiting for requests ...\n");
   fflush (stdout);
#endif
```

The *`CORBA_ORB_run()`* operation is then executed, starting the event loop which waits for incoming requests from clients. When the Log service is stopped the ORB's shutdown operation is called.

```
   CORBA_ORB_run (orb, &env);
   EORB_CHECK_EXC_RETURN_VAL ("ORB_run", &env, -1);

   CORBA_free (servant);

#if EORB_USE_STDIO
   fprintf (stdout, "LwLog example complete\n");
   fflush (stdout);
#endif

   return 0;
}
```

4.2 Client

The client code show here demonstrates how to create a Lightweight Log Service client.

Implementation

The client implementation file, *client.c*, imports the *LwLogService.h* header file, the same as included by the server, *server.c*. However, instead of including the *lwlogS.h* header file, the client includes the (IDL generated) *lwlog.h* file which contains the interface declarations needed by the client to connect, via the ORB and POA, to the Log server implementation.

```
#include "eOrbC/EORB/LwLogService.h"
#include "lwlog.h"
```

The *starttime* variable, of type CosLwLog_LogTime, is declared: starttime records the time, in seconds and nanoseconds, when a Log record is created. This variable will be used by the example for retrieving records based on their creation time stamps.

```
static CosLwLog_LogTime starttime;
```

The *printULongLong()* operation, included in the client code, merely accommodates the printing of non-ansi long long data types on Win32 platforms. This function is not needed for operating the Log service.

```
static void printULongLong (CORBA_unsigned_long_long ull)
{
#if EORB_USE_STDIO
#ifdef _WIN32
   fprintf (stdout, "%I64u\n", ull);
#else
   fprintf (stdout, "%lu\n", (unsigned long) ull);
#endif
   fflush (stdout);
#endif
}
```

The *printRecords()* operation demonstrates how to obtain information about a sequence of Log records by using *CORBA_sequence_CosLwLog_LogRecord*.

```
static void printRecords (CORBA_sequence_CosLwLog_LogRecord *
records)
{
   CORBA_unsigned_long i;

#if EORB_USE_STDIO
   fprintf (stdout, "\t  %d records recovered\n\n",
records->_length);
   fflush (stdout);

   for (i = 0; i < records->_length; i++)
   {
      CosLwLog_RecordId id = records->_buffer[i].id;
      CosLwLog_LogTime time = records->_buffer[i].time;
      CosLwLog_ProducerLogRecord info = records->_buffer[i].info;
```

```
      fprintf (stdout, "\t  RecordId      : ");
      printULongLong (id);
      fprintf (stdout, "\t  Time          : %d secs", time.seconds);
      fprintf (stdout, "  %d nsecs\n", time.nanoseconds);
      fprintf (stdout, "\t  Producer Id   : %s\n", info.producerId);
     fprintf (stdout, "\t  Producer Name : %s\n", info.producerName);
      fprintf (stdout, "\t  Log Level     : %d\n", info.level);
      fprintf (stdout, "\t  Log Data      : %s\n\n", info.logData);
      fflush (stdout);
   }
#endif
}
```

The *get_status()* operation demonstrates how to use *LogStatus* to obtain a Log's current status.

The operation successively calls

CosLwLog_LogStatus_get_max_size(), *CosLwLog_LogStatus_get_current_size()* and *CosLwLog_LogStatus_get_n_records()* operations.[1]

```
static void get_status (CORBA_Object status, CORBA_Environment * ev)
{
   CORBA_unsigned_long_long size;

   size = CosLwLog_LogStatus_get_max_size (status, ev);
   EORB_CHECK_EXC_RETURN ("LogStatus_get_max_size", ev);

#if EORB_USE_STDIO
   fprintf (stdout, "\t  Max Size is          : ");
   printULongLong (size);
#endif

   size = CosLwLog_LogStatus_get_current_size (status, ev);
   EORB_CHECK_EXC_RETURN ("LogStatus_get_current_size", ev);

#if EORB_USE_STDIO
   fprintf (stdout, "\t  Current Size is      : ");
   printULongLong (size);
#endif

   size = CosLwLog_LogStatus_get_n_records (status, ev);
   EORB_CHECK_EXC_RETURN ("LogStatus_get_n_records", ev);

#if EORB_USE_STDIO
   fprintf (stdout, "\t  Number of Records is : ");
   printULongLong (size);
   fflush (stdout);
#endif
}
```

1. See *interface LogStatus* on page 16 for a complete list of available `LogStatus` operations.

The *writeRecords()* operation demonstrates how *LogProducer* can be used to add a sequence of Log producer records to a Log.

After creating a sequence of producer Log records, *records*, and initialising the starttime variable which stores the current time (used by the example's *writeRecords()* operation), the operation creates a sequence of *CORBA_sequence_CosLwLog_ProducerLogRecord*s.

Each CosLwLog_ProducerLogRecord, *rec*, is added to the sequence of ProducerLogRecords after being assigned *producerId*, *producerName*, Log *level*, and *logData* values.

The completed ProducerLogRecords sequence is then written to the Log using the ProducerLogRecords::write_records() operation. The status of the Log is subsequently obtained and printed.

```
static void writeRecords (CORBA_Object producer, CORBA_Environment *
ev)
{
   CORBA_sequence_CosLwLog_ProducerLogRecord * records = NULL;
   CORBA_unsigned_long size = 4;
   CORBA_unsigned_long i;
   struct timespec now;

   records = CosLwLog_ProducerLogRecordSequence__alloc ();
   records->_buffer = CosLwLog_ProducerLogRecordSequence_allocbuf
(size);
   records->_length = size;
   CORBA_sequence_set_release (records, TRUE);

   clock_gettime (CLOCK_REALTIME, &now);

   starttime.seconds = now.tv_sec;
   starttime.nanoseconds = now.tv_nsec;

#if EORB_USE_STDIO
   fprintf (stdout, "\n\tWriting %d records\n\n", size);
   fflush (stdout);
#endif

   for (i = 0; i < size; i++)
   {
      CosLwLog_ProducerLogRecord rec;
      char id[8];
      char data[16];

      sprintf (id, "ID %d", i);
      sprintf (data, "Data Entry %d", i);

      rec.producerId = CORBA_string_dup (id);
      rec.producerName = CORBA_string_dup ("LwLog Example");
      rec.level = (CORBA_unsigned_short) i;
      rec.logData = CORBA_string_dup (data);

      records->_buffer[i] = rec;
   }
```

```
   CosLwLog_LogProducer_write_records (producer, records, ev);

   /* Clean up */

   CORBA_free (records);

   EORB_CHECK_EXC_RETURN ("LogProducer_write_records", ev);
}
```

The *retrieveRecords()* operation demonstrates how the *LogConsumer* interface can retrieve a sequence of Log producer records from a Log.

The variables used by the operation, as part of the retrieval process, are:

- *CORBA_sequence_CosLwLog_LogRecord records* - holds the retrieved records
- *CosLwLog_RecordId id* - holds the id value which will be used to identify the records to be retrieved
- *CORBA_unsigned_long howMany* - specifies the number of records which are to be retrieved by the operation

```
static void retrieveRecords (CORBA_Object consumer,
CORBA_Environment * ev)
{
   CORBA_sequence_CosLwLog_LogRecord * records = NULL;
   CosLwLog_RecordId id;
   CORBA_unsigned_long howMany;

   howMany = 4;

#if EORB_USE_STDIO
   fprintf (stdout, "\n\tretrieveRecords\n");
   fflush (stdout);
#endif

   id = CosLwLog_LogConsumer_get_record_id_from_time
   (
      consumer,
      &starttime,
      ev
   );
   EORB_CHECK_EXC_RETURN ("LogConsumer_get_record_id_from_time",
ev);

   records = CosLwLog_LogConsumer_retrieve_records
   (
      consumer,
      &id,
      &howMany,
      ev
   );
   EORB_CHECK_EXC_RETURN ("LogConsumer_retrieve_records", ev);

   printRecords (records);
```

```
   CORBA_free (records);
}
```

The *retrieveRecordsByProducerID()* operation shows how the *LogConsumer* interface can selectively retrieve a Log record using its producer id and the time the record was created.

```
static void retrieveRecordsByProducerId
(
   CORBA_Object consumer,
   CORBA_Environment * ev
)
{
   CORBA_sequence_CosLwLog_LogRecord * records = NULL;
   CosLwLog_RecordId id;
   CORBA_unsigned_long howMany;
   CosLwLog_StringSeq * vals = NULL;

#if EORB_USE_STDIO
   fprintf (stdout, "\n\tretrieveRecordsByProducerId\n");
   fflush (stdout);
#endif

   howMany = 4;

   vals = CosLwLog_StringSeq__alloc ();
   vals->_buffer = CosLwLog_StringSeq_allocbuf (1);
   vals->_buffer[0] = CORBA_string_dup ("ID 1");
   vals->_length = 1;
   CORBA_sequence_set_release (vals, TRUE);

   id = CosLwLog_LogConsumer_get_record_id_from_time (consumer,
&starttime, ev);
   EORB_CHECK_EXC_RETURN ("LogConsumer_get_record_id_from_time",
ev);

   records = CosLwLog_LogConsumer_retrieve_records_by_producer_id
   (
      consumer,
      &id,
      &howMany,
      vals,
      ev
   );
   EORB_CHECK_EXC_RETURN
("LogConsumer_retrieve_records_by_producer_id", ev);

   printRecords (records);

   CORBA_free (records);
   CORBA_free (vals);
}
```

The client's main function is implemented using the Spectra ORB *EORB_MAIN* macro. The main function performs initialisations and declarations (including the ORB and Log objects. Main then runs the operations, described above, which demonstrate the various Log operations.

```
EORB_MAIN (client)
{
   CORBA_Environment env;
   CORBA_Object server;
   CORBA_Object log;
   CORBA_ORB orb;

   /* Hook in the IIOP profile */

   EORB_IIOP_plugin ();

#if EORB_USE_STDIO
   fprintf (stdout, "LwLog example starting\n");
   fflush (stdout);
#endif

   orb = CORBA_ORB_init (&argc, argv, "eorb-ce", &env);
   EORB_CHECK_EXC_RETURN_VAL ("CORBA_ORB_init", &env, -1);

   /* Resolver servers */

   log = CORBA_ORB_resolve_initial_references (orb, "LogService",
&env);
   EORB_CHECK_EXC_RETURN_VAL ("resolve_initial_references", &env,
-1);

   server = CORBA_ORB_resolve_initial_references (orb, "server",
&env);
   EORB_CHECK_EXC_RETURN_VAL ("resolve_initial_references", &env,
-1);

   /* Call to server */

#if EORB_USE_STDIO
   fprintf (stdout, "\tDefault status\n");
   fflush (stdout);
#endif
   get_status (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("get_status", &env, -1);

   writeRecords (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("writeRecords", &env, -1);

#if EORB_USE_STDIO
   fprintf (stdout, "\tCurrent status\n");
   fflush (stdout);
#endif
   get_status (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("get_status", &env, -1);

   retrieveRecords (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("retrieveRecords", &env, -1);
```

```
   retrieveRecordsByProducerId (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("retrieveRecordsByProducerId", &env,
-1);

   CosLwLog_LogAdministrator_clear_log (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("LogAdministrator_clear_log", &env,
-1);

#if EORB_USE_STDIO
   fprintf (stdout, "\n\n\tFinal status\n");
   fflush (stdout);
#endif
   get_status (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("get_status", &env, -1);

   CosLwLog_LogAdministrator_destroy (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("LogAdministrator_destroy", &env, -1);

   /* Shutdown server */

   lwlog_shutdown (server, &env);
   EORB_CHECK_EXC_RETURN_VAL ("lwlog_shutdown", &env, -1);

   /* Clean up */

   CORBA_Object_release (log, &env);
   EORB_CHECK_EXC_RETURN_VAL ("CORBA_Object_release", &env, -1);
   CORBA_Object_release (server, &env);
   EORB_CHECK_EXC_RETURN_VAL ("CORBA_Object_release", &env, -1);

   CORBA_ORB_destroy (orb, &env);
   EORB_CHECK_EXC_RETURN_VAL ("CORBA_ORB_destroy", &env, -1);

#if EORB_USE_STDIO
   fprintf (stdout, "LwLog example complete\n");
   fflush (stdout);
#endif

   return 0;
}
```

Index

Index

R

S

W