Lightweight Log Service Specification

November 2002
Draft Adopted Specification
ptc/02-11-02

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;

- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;

- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and

- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/ .

OMG Documents

The OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

The OMG documentation is organized as follows:

OMG Modeling Specifications

Includes the UML, MOF, XMI, and CWM specifications.

OMG Middleware Specifications

Includes CORBA/IIOP, IDL/Language Mappings, Specialized CORBA specifications, and CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

Includes CORBAservices, CORBAfacilities, OMG Domain specifications, OMG Embedded Intelligence specifications, and OMG Security specifications.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.) OMG formal documents are available from our web site in PostScript and PDF format. Contact the Object Management Group, Inc. at:

<div align="center">

OMG Headquarters

250 First Avenue

Needham, MA 02494

USA

Tel: +1-781-444-0404

Fax: +1-781-444-0320

pubs@omg.org

http://www.omg.org

</div>

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted and/or supported parts of this specification:
- 88solutions Corporation
- BAE
- Mercury Computer Systems
- MITRE Corporation
- Raytheon Company
- Rockwell Collins

Overview *1*

Contents

This chapter contains the following sections.

Section Title	Page
"Scope"	1-1
"Purpose"	1-2
"Relationship to the Realtime, Embedded, and Specialized Systems Platform Taskforce"	1-2
"Relation to Existing OMG Specifications"	1-2
"Relation to Pending OMG Specifications"	1-3
"Compliance"	1-3

1.1 Scope

The Software Communications Architecture (SCA) is an open architecture defined by the Joint Tactical Radio Systems (JTRS) Joint Program Office (JPO). The SCA defines a framework based on CORBA middleware layered on top of a real-time operating system. The framework provides the common infrastructure for the software defining the characteristics of the software-defined radio system. One component of this framework is a central logging facility, enabling the asynchronous collection of informational messages from any component connected to the framework; and the controlled read access to this information. The transition of the particular section of the SCA specification defining the CORBA-based logging facility into an OMG specification is the scope of this document.

1

In addition, the CORBA-based specification, proven by several implementations, has been abstracted into Platform Independent Model, expressed in UML, to make this service available to a wider audience. Intended are primarily embedded and/or real-time usages. However, other areas might equally benefit from using this service.

1.2 Purpose

The Lightweight Logging Service specification contained in this document is primarily intended as an efficient, central facility inside an embedded or real-time environment to accept and manage logging records. These records are emitted from applications residing in the same environment and stored in a memory-only storage area owned and managed by the Lightweight Logging Service. The service was designed to be a mostly compatible subset of the Telecom Log Service, however, it differs in the way logging records are written to the log; or looked up and retrieved from the log.

This service has a much wider application than just the software-defined radio domain. It will find its way into all areas of embedded systems, like machine control, onboard vehicle systems, etc., but also into ubiquitous computing devices like pocket computer and electronic organizers. But also "regular" application areas will benefit, if just a small, memory-only logging facility is required.

1.3 Relationship to the Realtime, Embedded, and Specialized Systems Platform Taskforce

The Lightweight Logging Service RFC fits the plans of the Realtime, Embedded, and Specialized Systems Platform Taskforce by filling the first slot in a planned series of lightweight service specifications.

1.4 Relation to Existing OMG Specifications

The Lightweight Logging Service described in this document was in principle designed as a subset of the Telecom Log Service targeted for embedded and/or real-time systems. However, the severe resource constraints typically present in those systems required several deviations from the Telecom Log Service specification. The following list shows the major deviations:

- Only simple logging is supported. The Lightweight Logging Service is a stand-alone service targeted for use in resource constraint embedded and/or real-time environemts. The Lightweight Logging Service does not inherit from the Event- or Notifcation Service.

- The Lightweight Logging Service provides no connection to event cannels of any kind. Instead log producer and consumer interfaces are provided.

- The Lightweight Logging Service does not support any federation of logging services. In particular, the Lightweight Logging Service does not support forwarding of log records to another Log object implementation

- Logging information is only stored in memory. No persistent log record store is supported.

- No filters are supported.

- Due to the constraints of an embedded environment, the Lightweight Logging Service uses a dedicated structure to hold logging records, instead of type any in the Telecom Log Service. Combined with a list of "well known" typecodes, this structure provides the restrictive control on type variety necessary in an embedded system. Further, an "any-less" structure simplifies the use of embedded ORBs, which frequently impose restrictions on type any.

- The only query-like operation is a lookup by time for a single record; otherwise read access to a record is only via its record ID.

- The Lightweight Logging Service provides the service to read logging records in a series of small consecutive chunks, very similar to the use of an iterator.

- Write operations to the Log are strictly asynchronous. The log provides no feedback or exceptions; this would interfere with the timing constraints of the log producer.

1.5 Relation to Pending OMG Specifications

The Lightweight Logging Service addresses primarily the logging needs of embedded systems, which may or may not have real-time behavior. Therefore it relies only on a minimum set of system resources. A platform specific CORBA-based implementation would only require capabilities slightly above the Minimum CORBA Profile, a capability set currently under discussion as "Embedded CORBA Profile" in the Real-Time and Embedded Methods Platform Taskforce.

This specification is also fully aligned with the current planning of the Real-Time and Embedded Methods Platform Taskforce to create a set of lightweight services targeted for the embedded CORBA market.

1.6 Compliance

This specification consists of two parts, a Platform Independent Model (PIM) and a Platform Specific Model (PSM), specifying a realization of the PIM in the terms of CORBA IDL. Both parts each represent indivisible pieces of work. Conformant implementations must either provide an implementation which represents a complete mapping of the PIM into the selected target technology; or it must provide a complete implementation of the CORBA IDL PSM described in this document.

No partial implementation of either the PIM or the PSM is deemed conformant.

Platform Independent Model 2

Contents

This chapter contains the following sections.

2.1 Overview and Architecture

In consideration of the resource constraints imposed by the embedded system environment, the Lightweight Logging Service is a free-standing, self-contained service, and not connected to an event channel or similar infrastructure. The core of the Lightweight Logging Service is represented by the class **Log**, which encapsulates the storage area for logging records and provides the methods comprising the logging functionality. However, the class **Log** does not communicate directly with the rest of the environment. Communication with the surrounding environment is handled through three distinct interfaces.

LogProducer This interface allows the insertion of new log records into the logging storage area encapsulated by the Log class. In favor of preserving the overall operational integrity of the system, no guarantee is made that a logging record is accepted and stored if the logging service is unable to process and /or store it.

LogConsumer This interface allows the retrieval of logging records from the storage area encapsulated by the Log class.

LogAdmin This interface provides the management functionality to operate and manage the logging service.

The above three interfaces are derived from an abstract super interface **LogStatus**, which provides informational functionality common to all three interfaces.
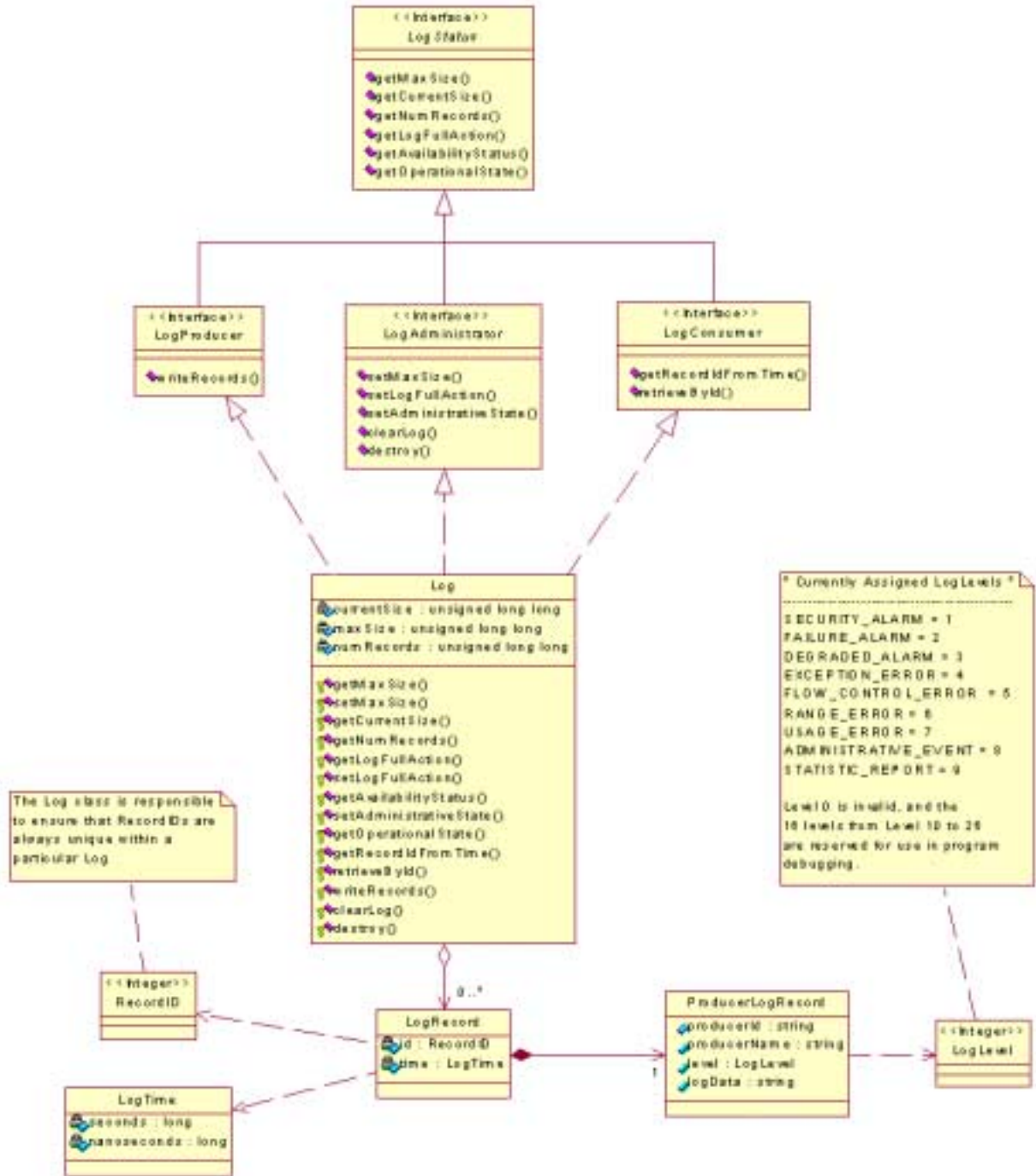
Figure 2-1   Lightweight Logging Service PIM

As shown in Figure 2-1, the central piece of the Lightweight Logging Service is the class Log, which encapsulates the storage area for logging records and provides all necessary operations to manage and operate the Lightweight Logging Service. Note, however, that the operations should not be directly accessible to any clients of the logging service. Instead, a set of interfaces is provided to give controlled access to each kind of clients. This is kind of a "poor man's" protection system, which provides sufficient protection against accidental misuse, while, at the same time, giving tribute to the severe resource constraints common in embedded devices.

2.2 Type Definitions

2.2.1 InvalidParam Exception

The InvalidParam exception indicates that a provided parameter was invalid. Details about the cause for this exception are delivered in the string attribute **details**.

```
<<exception>>
 InvalidParam
details : string
```
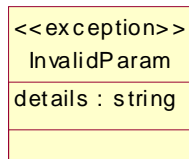
2.2.2 LogLevel

The LogLevel allows a classification of the logging record. The value provided is recorded in the logging record and provided to the consumer at retrieval, but it has no particular meaning or side effects during storage of the record in the Log.

```
<<Integer>>
 LogLevel
```
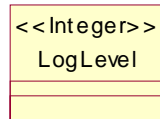
The implementation of the **LogLevel** type should provide a mechanism to assign the following values, or an equivalent implementation thereof, to an instance of the **LogLevel** type.

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**

Further, an implementation should reserve the codes 10-26, or their equivalent, do denote program debugging messages.

2.2.3 OperationalState

The enumeration **OperationalState** defines the Log states of operation. When the Log is **ENABLED** it is fully functional and available for use by log producer and log consumer clients. A Log that is **DISABLED** has encountered a runtime problem and is not available for use by log producers or log consumers. The internal error conditions that cause the Log to go into **DISABLED** state are implementation specific.
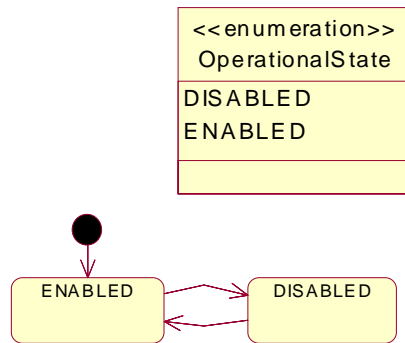
2.2.4 AdministrativeState

The **AdministrativeState** denotes the active logging state of an operational Log. When set to **UNLOCKED** the Log will accept records for storage, per its operational parameters. When set to **LOCKED** the Log will not accept new log records and records can be read or deleted only.
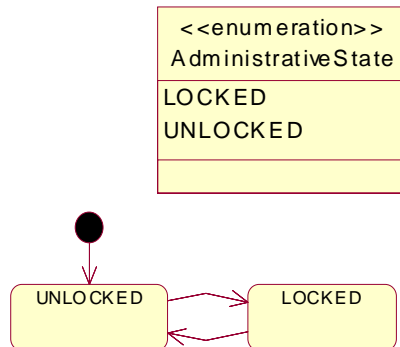
2.2.5 AvailabilityStatus

The **AvailabilityStatus** denotes whether or not the Log is available for use. When true, **offDuty** indicates the Log is **LOCKED** (administrative state) or **DISABLED** (operational state). When true, **logFull** indicates the Log storage is full.

```
+-----------------------+
|  Availability Status  |
+-----------------------+
| offDuty : boolean     |
| logFull : boolean     |
+-----------------------+
|                       |
+-----------------------+
```
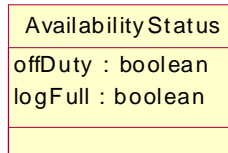
2.2.6 LogFullAction

This type specifies the action that the Log should take when its internal buffers become full of data, leaving no room for new records to be written. **WRAP** indicates that the Log will overwrite the oldest LogRecords with the newest records, as they are written to the Log. **HALT** indicates that the Log will stop logging when full.

```
+-----------------------+
|  <<enumeration>>      |
|   LogFullAction       |
+-----------------------+
| WRAP                  |
| HALT                  |
+-----------------------+
|                       |
+-----------------------+
```
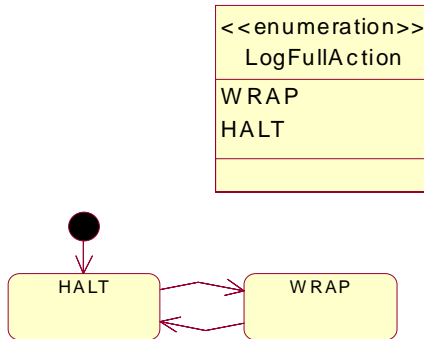

2.2.7 RecordID

This type provides the record ID that is assigned to a **LogRecord** by the **Log**; the **RecordID** must be unique.

This type should be able to hold a 64 bit integer quantity or equivalent

```
+-----------------------+
|  <<Integer>>          |
|   RecordID            |
+-----------------------+
|                       |
+-----------------------+
|                       |
+-----------------------+
```
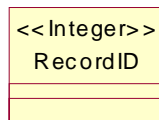
2.2.8 LogTime

This type provides the time format used by the Log to time stamp **LogRecords**. The fields of this type are intentionally designed to map directly to the POSIX timespec structure.

LogTime
seconds : long nanoseconds : long

2.2.9 LogRecord

The **LogRecordType** defines the format of the LogRecords as stored in the Log. It represents an encapsulation of the **ProducerLogRecord**, supplied by the log producer, and adds the time stamp (via the **LogTime** structure) and a unique record identification (via the **RecordID** field). Refer to Figure 2-2.

2.2.10 LogRecordSequence

The **LogRecordSequence** type defines an unbounded sequence of **LogRecords**. Refer to Figure 2-2.

2.2.11 ProducerLogRecord

The **ProducerLogRecord** represents the log record written by the log producer client to the log. It will be encapsulated by in a **LogRecord** object before it is stored in the log storage area. Refer to Figure 2-2.
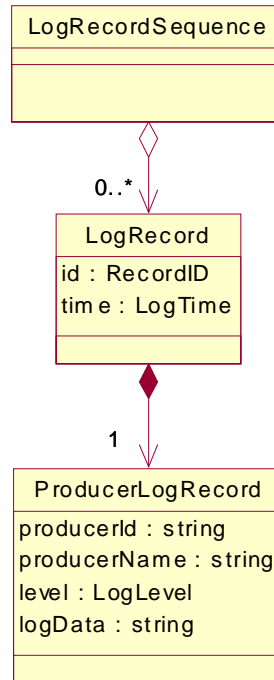
Figure 2-2 LogRecordSequence, LogRecord, and ProducerLogRecord

2.3 Common Interface Operations

Interface **LogStatus** provides access to operations of common interest, which are through inheritance available in all interfaces of the logging service.

2.3.1 getMaxSize

Returns the size of the logging storage area.

Synopsis
```
+ getMaxSize () : unsigned long long
```

Parameters and Return

Parameter	Type	Description
`<return>`	`unsigned long long`	The maximum size of the log storage area in bytes

Exceptions

This function raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The available space in this storage area is finite. This operation returns the maximum capacity in bytes of the storage area.

2.3.2 getCurrentSize

Returns the amount of log storage area currently occupied by logging records.

Synopsis

+ getCurrentSize () : unsigned long long

Parameters and Return

Parameter	Type	Description
`<return>`	`unsigned long long`	The size of the currently used log storage area in bytes

Exceptions

This function raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The **getCurrentSize** operation returns the size in bytes of the log storage area currently occupied by logging records. This value is less or equal to the total storage area size returned by the **getMaxSize** operation.

2.3.3 getNumRecords

Returns the number of records presently stored in the Log.

Synopsis

+ getNumRecords () : unsigned long long

Parameters and Return

Parameter	Type	Description
<return>	**unsigned long long**	The number of logging records currently stored in the storage area.

Exceptions

This operation raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The **getNumRecords** operation returns the number of logging records currently stored in the log storage area.

2.3.4 getLogFullAction

Returns the action to be taken when the storage area becomes full.

Synopsis

+ getLogFullAction () : LogFullAction

Parameters and Return

Parameter	Type	Description
<return>	**LogFullAction**	The selected alternative of the LogFullAction enumeration.

Exceptions

This operation raises no exceptions.

Description

Since the storage space of the Log storage area is finite, the Logging Service has to take special action when the free space is depleted. The kind of action is described by the **LogFullAction** type. The **getLogFullAction** operation returns the information about which action the Logging Service will take when the storage area becomes full. The possible values are **HALT**, which means no further logging records are accepted and stored; or **WRAP**, which means the log continues by overwriting the oldest records in the storage area.

2.3.5 getAvailabilityStatus

Returns the availability status of the Log.

Synopsis

+ getAvailabilityStatus () : AvailabilityStatus

Parameters and Return

Parameter	Type	Description
<return>	**AvailabilityStatus**	An instance of the AvailabilityStatus representing the actual status of the log.

Exceptions

This operation raises no exceptions.

Description

The ability of the Log to accept and store logging records might become impaired. The **getAvailabilityStatus** operation is used to check the availability status of the Log. The returned instance of the **AvailibilityStatus** type contains two Boolean values: **offDuty**, which indicates the log is disabled when true, and **logFull**, which indicates that all free space is depleted in the log storage area.

2.3.6 getAdministrativeState

Returns the administrative state of the Log.

Synopsis

+ getAdministrativeState () : AdministrativeState

Parameters and Return

Parameter	Type	Description
<return>	**AdministrativeState**	The actually selected alternative of the AdministrativeState enumeration.

Exceptions

This operation raises no exceptions.

Description

The ability of the logging service to accept and store new logging records can be affected by administrative action. The **getAdministrativeState** is used to read the administrative state of the Log. The possible states are **LOCKED** and **UNLOCKED**. If the state is **LOCKED**, no new records are accepted. Reading of already stored records is not affected.

2.3.7 getOperationalState

Returns the operational state of the Log.

Synopsis

+ getOperationalState () : OperationalState

Parameters and Return

Parameter	Type	Description
<return>	**OperationalState**	The actually selected alternative of the OperationalState enumeration.

Exceptions

This operation raises no exceptions.

Description

The **getOperationalState** operation returns the actual operational state of the log. Possible values are **ENABLED**, which means the log is fully functional and available to log producer and log consumer clients; or **DISABLED**, which indicates the log has encountered a runtime problem and is not available for use by log producers or log consumers.

2.4 LogConsumer Interface Operations

2.4.1 getRecordIdFromTime

Identify a record in the log a record based on its time stamp.

Synopsis

+ getRecordIdFromTime (in fromTime : LogTime)
**   : RecordID**

Parameters and Return

Parameter	Type	Description
`fromTime`	`LogTime`	The timestamp to start the search with
`<return>`	`RecordID`	Record ID of the first record matching the timestamp.

Exceptions

This operation raises the InvalidParam exception if the supplied parameter is invalid.

Description

The `getRecordIdFromTime` operation returns the record Id of the first record in the Log with 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 the criteria provided, then the **RecordId** returned corresponds to the next record that will be recorded in the future. In this way, if this "future" `recordId` is passed into the `retrieveById` operation, an empty record will be returned unless records have been recorded since the time specified. Note that if the time specified in the `fromTime` parameter is in the future, there is no guarantee that the resulting records returned by the `retrieveById` operation will have a time stamp after the `fromTime` parameter if the returned `recordId` from this invocation of the `getRecordIdFromTime` operation is subsequently used as input to the `retrieveById` operation.

2.4.2 retrieveById

Retrieves a specified number of records from the Log.

Synopsis

```
+ retrieveById (inout currentId  : RecordID,
                 inout howMany   : unsigned long)
                                 : LogRecordSequence
```

Parameters and Return

Parameter	Type	Description
`currentId`	`RecordId`	The ID of the starting record
`howMany`	`unsigned long`	The number of records to retrieve, will be updated to the number of records actually retrieved.
`<return>`	`LogRecordSequence`	The sequence of retrieved records

Exceptions

This operation raises the InvalidParam exception if the supplied parameter is invalid.

Description

The `retrieveById` operation returns a **LogRecordSequence** that begins with the record specified by the `currentId` parameter. The number of records in the **LogRecordSequence** returned by the `retrieveById` operation is equal to the number of records specified by the `howMany` parameter, or the number of records available if the number of records specified by the `howMany` parameter cannot be met. The log will update `howMany` to indicate the number of records returned and will set `currentId` to the id of the record following the last retrieved record. If there are no further records available, `currentId` will be set to zero. If the record specified by `currentId` does not exist, or if the Log is empty, the `retrieveById` operation returns an empty list of **LogRecords**, and sets both, `currentId` and `howMany` to zero.

2.5 LogProducer Interface Operations

2.5.1 writeRecords

Writes records to the Log.

Synopsis

```
+ writeRecords (in records : ProducerLogRecordSequence)
                              : void
```

Parameters and Return

Parameter	Type	Description
records	**ProducerLogRecordSequence**	The records to be written to the log.
<return>	**void**	This operation does not return a value.

Exceptions

This operation raises no exceptions.

Description

The **writeRecords** operation adds the log records supplied in the **records** parameter to the Log. When there is insufficient storage to add one of the supplied log records to the Log, and the **LogFullAction** is set to **HALT**, the **writeRecords** operation shall set the availability status logFull state to true. For example, if 3 records are provided in the records parameter, and while trying to write the second record to the log, the record will not fit, then the log is considered to be full. Therefore, the second and third records will not be stored in the log but the first record would have been successfully stored.

The **writeRecords** operation inserts the current UTC time to the **time** field of each record written to the Log, and assigns a unique **record Id** to the **id** field of the **LogRecord**.

Log records accepted for storage by the **writeRecords** will be available for retrieval in the order received.

2.6   LogAdministrator Interface Operations

2.6.1   setMaxSize

Sets the maximum size the Log storage area.

Synopsis

+ setMaxSize(in size : unsigned long long) : void

Parameters and Return

Parameter	Type	Description
`size`	`unsigned long long`	The desired size for the logging storage area in bytes
`<return>`	`void`	This operation does not return a value

Exceptions

This operation raises the InvalidParam exception if the supplied parameter is invalid.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The available space in this storage area is finite. This operation allows setting of the maximum capacity in bytes of the storage area. Note, however, that this operation might be constrained by the underlying operation (you can't assign more memory than is physically present), or a platform specific implementation might decide to render this operation as a no-op and provide a fixed maximum size instead.

2.6.2 setLogFullAction

Configure the action to be taken if the log storage area becomes full.

Synopsis

`+ setLogFullAction (in action : LogFullAction) : void`

Parameters and Return

Parameter	Type	Description
`action`	`LogFullAction`	Specify the desired selection from the LogFullAction enumeration (either HALT or WRAP)
`<return>`	`Void`	

Exceptions

This operation raises no exceptions.

Description

Since the storage space of the Log storage area is finite, the Logging Service has to take special action when the free space is depleted. The kind of action is described by the **LogFullAction** type. The **setLogFullAction** operation allows the specification of

actions which should be taken after all free space in the log storage area is depleted. The possible values are **HALT**, which means no further logging records are accepted and stored; or **WRAP**, which means the log continues by overwriting the oldest records in the storage area.

2.6.3 setAdministrativeState

This operation provides write access to the administrative state value.

Synopsis

```
+ setAdministrativeState (in state : AdministrativeState)
                         : void
```

Parameters and Return

Parameter	Type	Description
state	**AdministrativeState**	Select the desired alternative from the AdministrativeState enumeration. (Possible values are LOCKED and UNLOCKED.)
<return>	**Void**	

Exceptions

This operation raises no exceptions.

Description

This operation affects the ability of the logging service to accept and store new logging records by administrative action. The possible states are **LOCKED** and **UNLOCKED**. If the state is **LOCKED**, no new records are accepted. Reading of already stored records is not affected. If the state is set to **UNLOCKED**, the log operates normally.

2.6.4 clearLog

Purge the log storage area.

Synopsis

```
clearLog () : void
```

Parameters and Return

This operation has no parameters or returns.

Exceptions

This operation raises no exceptions.

Description

This operation purges all logging records from the log storage area; however, it does not alter the size of the storage area in any way. A subsequent invocation of the **getCurrentSize** operation will return 0 (zero).

2.6.5 destroy

Tear down an instantiated Log.

Synopsis

Destroy () : void

Parameters and Return

This operation has no parameters or returns.

Exceptions

This operation raises no exceptions.

Description

This operation will destroy the associated instance of the **Log** class. All existing records in the log storage area are irrecoverably lost and the memory resources associated with the storage area are released.

Platform Specific Model: Mapping to CORBA IDL   *3*

Contents

This chapter contains the following sections.

3.1 Overview

This specification defines a Lightweight Logging Service intended for use in resource-constraint systems like embedded and/or real-time CORBA systems. It represents the CORBA Platform Specific Model (PSM) derived from the Lightweight Logging Service Platform Independent Model (PIM) described in Chapter 2 of this document. In this particular case, this PSM is "the original," derived from the Software Communication Architecture (SCA) version 2.2. SCA defines the system platform for software-defined radios, using CORBA on top of a real-time operating system. The PIM described in Chapter 2 was derived from this PSM through generalization.

3.1.1 Mapping from the Platform Independent Model

The mapping between the elements of the Platform Independent Model described in Chapter 2 and the corresponding elements of the CORBA IDL Platform Specific Model described in the following sections is in most cases one-to-one. A note in the description of each PSM element will explain the correspondence between the PSM element and its counterpart in the PIM.

According to the characteristic of CORBA to fully encapsulate the object underlying the provided interfaces, no visible mapping exists between the UML **Log** class of the PIM and a CORBA IDL construct in the PSM. The operations to be implemented by the underlying **Log** object are only visible through the four interfaces, as defined in the PIM, and are fully described in the corresponding interface sections below.

3.2 Types and Data Structures

3.2.1 LogLevel

Type **LogLevel** is an enumeration-like type that is utilized to identify log levels.

```
unsigned short LogLevel;

const unsigned short SECURITY_ALARM = 1;
const unsigned short FAILURE_ALARM = 2;
const unsigned short DEGRADED_ALARM =3;
const unsigned short EXCEPTION_ERROR =4;
const unsigned short FLOW_CONTROL_ERROR =5;
const unsigned short RANGE_ERROR =6;
const unsigned short USAGE_ERROR = 7;
const unsigned short ADMINISTRATIVE_EVENT = 8;
const unsigned short STATISTIC_REPORT = 9;
// Values ranging from 10 to 26 are reserved for
// 16 debugging levels.
```

The **LogLevel** allows a classification of the logging record. The value provided is recorded in the logging record and provided to the consumer at retrieval, but it has no particular meaning or side effects during storage of the record in the Log.

Mapping from the Platform Independent Model

This IDL integer type is the result of a one-to-one mapping from the UML classifier LogLevel (stereotyped as <<Integer>>), described in Section 2.2.2, "LogLevel," on page 2-4. Note that the first 27 values (from 0 to 26) are predefined by the PIM.

Difference to the Telecom Log Service

This type does not exist in the Telecom Log Service.

3.2.2 OperationalState

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

The enumeration **OperationalStateType** defines the Log states of operation. When the Log is **enabled** it is fully functional and is available for use by log producer and log consumer clients. A Log that is **disabled** has encountered a runtime problem

and is not available for use by log producers or log consumers. The internal error conditions that cause the Log to set the operational state to **enabled** or **disabled** are implementation specific.

Mapping from the Platform Independent Model

This IDL enumeration type is the result of a one-to-one mapping from the UML classifier **OperationalState** (stereotyped as **<<enumeration>>**), described in Section 2.2.3, "OperationalState," on page 2-5. The identifiers of the enumeration values have been converted to lower-case for compatibility with the Telecom Log Service.

Difference to the Telecom Log Service

This IDL enumeration type is identical to the corresponding type definition in the Telecom Log Service.

3.2.3 AdministrativeState

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

The AdministrativeStateType denotes the active logging state of an operational Log. When set to **unlocked** the Log will accept records for storage, per its operational parameters. When set to **locked** the Log will not accept new log records and records can be read or deleted only.

Mapping from the Platform Independent Model

This IDL enumeration type is the result of a one-to-one mapping from the UML classifier **AdministrativeState** (stereotyped as **<<enumeration>>**), described in Section 2.2.4, "AdministrativeState," on page 2-5. The identifiers of the enumeration values have been converted to lower-case for compatibility with the Telecom Log Service.

Difference to the Telecom Log Service

This IDL enumeration type is identical to the corresponding type definition in the Telecom Log Service.

3.2.4 LogFullAction

```
enum LogFullAction (WRAP, HALT);
```

This type specifies the action that the Log should take when its internal buffers become full of data, leaving no room for new records to be written. **WRAP** indicates that the Log will overwrite the oldest **LogRecords** with the newest records, as they are written to the Log. **HALT** indicates that the Log will stop logging when full.

Mapping from the Platform Independent Model

This IDL enumeration type is the result of a one-to-one mapping from the UML classifier **LogFullAction** (stereotyped as **<<enumeration>>**), described in Section 2.2.6, "LogFullAction," on page 2-6.

Difference to the Telecom Log Service

The open-ended list of short integer values in the Telecom Log Service has been replaced by a two-element enumeration to better accommodate the constraints of the embedded environment. The enumeration values are retained in upper-case to distinguish from the constants used by the Telecom Log Service.

3.2.5 LogAvailabilityStatus

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

The **AvailabilityStatus** denotes whether or not the Log is available for use. When **true**, **off_duty** indicates the Log is **locked** (administrative state) or **disabled** (operational state). When **true**, **log_full** indicates the Log storage is full.

Struct member	Description
off_duty	Indicates that the log is unavailable, if true.
log_full	Indicates that the log storage area is full, if true.

Mapping from the Platform Independent Model

This IDL structure type is the result of a one-to-one mapping from the UML class **AvailabilityStatus**, described in Section 2.2.5, "AvailabilityStatus," on page 2-6.

3.2.6 LogTime

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

This type provides the time format used by the Log to time stamp LogRecords. Each field is intended to directly map to the POSIX timespec structure.

Note – An implementation should exclusively use UTC for time recording to support location transparency.

Mapping from the Platform Independent Model

This IDL structure type is the result of a one-to-one mapping from the UML class **LogTime**, described in Section 2.2.8, "LogTime," on page 2-7.

Difference to the Telecom Log Service

The **LogTime** structure replaces the use of the Time Service **TimeT** type. This way the dependency on the Time Service has been eliminated and the time specification aligned with the POSIX **timespec** structure, which is implemented by virtually all existing operating systems for embedded systems.

3.2.7 ProducerLogRecord

```
struct ProducerLogRecord {
    string    producerId;
    string    producerName;
    LogLevel  level;
    string    logData;
};
typedef sequence <ProducerLogRecordType>
                          ProducerLogRecordSequence;
```

Log producers format log records as defined in the structure **ProducerLogRecord**.

Struct member	Description
producerId	This field uniquely identifies the source of a log record. The value is the component's identifier and is unique for each SCA *Resource* and Core Framework component within the Domain.
producerName	This field identifies the producer of a log record in textual format. This field is assigned by the log producer, thus is not unique within the Domain (e.g., multiple instances of an application will assign the same name to the ProducerName field.)
level	The level field can be used to classify the log record according to the LogLevel type.
logData	This field contains the informational message being logged.

This structure represents a logging record written by a log producer client to the Log via the **LogProducer** interface. Upon reception, it is encapsulated by the **LogRecord** described in Section 3.2.1, "LogLevel," on page 3-2.

Mapping from the Platform Independent Model

This IDL structure type is the result of a one-to-one mapping from the UML class **ProducerLogRecord**, described in Section 2.2.11, "ProducerLogRecord," on page 2-7.

Difference to the Telecom Log Service

The **ProducerLogRecord** structure replaces the use of the IDL **any** type in the **LogRecord** of the Telecom Log Service. This is required in lieu of the lightweight nature of this service; and by the fact that many embedded ORB implementations do not support type **any**.

3.2.8 RecordId

```
typedef unsigned long long RecordId;
```

This type provides the unique record ID that is assigned to a **LogRecord** by the **Log**.

Mapping from the Platform Independent Model

This IDL type is the result of a one-to-one mapping from the UML classifier RecordID, as described in Section 2.2.7, "RecordID," on page 2-6. Defined as an unsigned long long it is capable to hold a 64 bit integer value, as required by the PIM.

Difference to the Telecom Log Service

The type RecordId is identical to the type used in the Telecom Log Service for simple log records.

3.2.9 LogRecord

```
struct LogRecord {
    RecordId         id;
    LogTime          time;
    ProducerLogRecord info;
};
```

```
typedef sequence<LogRecordType> LogRecordSequence;
```

The **LogRecord** type defines the format of the log records as stored in the **Log**. The 'info' field is the **ProducerLogRecord** that is written by a producer client to the Log.

The **LogRecordSequence** type defines an unbounded sequence of **LogRecords**.

Mapping from the Platform Independent Model

This IDL structure type is the result of a one-to-one mapping from the UML classes **LogRecord** and **LogRecordSequence**, described in Section 2.2.9, "LogRecord," on page 2-7 and Section 2.2.10, "LogRecordSequence," on page 2-7, and the aggregation between these classes.

Difference to the Telecom Log Service

The **LogRecord** structure was loosely modeled after the Telecom Log Service
LogRecord structure. However, since many embedded ORBs are not supporting the IDL
type **any**, the **ProducerLogRecord** structure replaces the any-typed info field in the
Telecom Log Service **LogRecord**. Further, the Lightweight Logging Service does not
support attributes in **LogRecords**.

Struct member	Description
Id	This field uniquely identifies a log record in the Log.
Time	This field holds the timestamp for the record.
Info	This field contains the logging record supplied by the producer.

3.3 Logging Interfaces

Operations on the Log object are separated into three distinct concrete interfaces. Each of
these interfaces represents a different access kind or privilege. This represents a
lightweight method of protection for the underlying Log object, without adding any
additional code. For the typically severe resource constrained embedded environments
this Lightweight Logging Service is addressing, the code saving is important, and the
protection functionality is considered sufficient.

Difference to the Telecom Log Service

The way the Lightweight Logging service is integrated into the surrounding environment
is very different from the Telecom Log Service (which is based on Event- or Notification
Channels). The Lightweight Logging Service is a stand-alone service targeted for
embedded systems, where the variety of client applications is limited and usually well-
known. The specified interfaces aim for a minimum footprint.

3.3.1 Interface LogStatus

```
interface LogStatus {
   unsigned long long get_max_size();
   unsigned long long get_current_size();
   unsigned long long get_n_records();
   LogFullAction get_log_full_action();
   AvailabilityStatus get_availability_status();
   AdministrativeState get_administrative_state();
   OperationalState get_operational_state();
};
```

The purpose of this interface is to make common operations equally available in the three
concrete interfaces inherited form this interface. These operations provide a common and
consistent way to query the actual state of a **Log** object. No state changes are permitted
or implied through the operations offered in this interface.

From a client's perspective, this interface should be considered as abstract; its operations should be invoked only in the context of the inherited interfaces.

3.3.1.1 *get_max_size*

Returns the size of the logging storage area.

Parameters and Return

Parameter	Type	Description
\<return\>	**unsigned long long**	The maximum size of the log storage area in bytes

Exceptions

This function raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The available space in this storage area is finite. This operation returns the maximum capacity in bytes of the storage area.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **getMaxSize**, defined in the UML class **Log** and made visible through interface **LogStatus** in the PIM (See Section 2.3.1, "getMaxSize," on page 2-8). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name, signature and result to the equivalent operation of the Telecom Log Service.

3.3.1.2 *get_current_size*

Returns the amount of log storage area currently occupied by logging records.

Parameters and Return

Parameter	Type	Description
`<return>`	`unsigned long long`	The size of the currently used log storage area in bytes.

Exceptions

This function raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The `get_current_size` operation returns the size in bytes of the log storage area currently occupied by logging records. This value is less or equal to the total storage area size returned by the `get_max_size` operation.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **getCurrentSize**, defined in the UML class **Log** and made visible through interface **LogStatus** in the PIM (See Section 2.3.2, "getCurrentSize," on page 2-9). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name, signature and result to the equivalent operation of the Telecom Log Service.

3.3.1.3 *get_n_records*

Returns the number of records presently stored in the Log.

Parameters and Return

Parameter	Type	Description
`<return>`	`unsigned long long`	The number of logging records currently stored in the storage area.

Exceptions

This operation raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The **get_n_records** operation returns the number of logging records currently stored in the log storage area.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **getNumRecords**, defined in the UML class **Log** and made visible through interface **LogStatus** in the PIM (See Section 2.3.3, "getNumRecords," on page 2-9). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide; and for compatibility with the Telecom Log Service.

Difference to the Telecom Log Service

This operation is identical in name, signature and result to the equivalent operation of the Telecom Log Service.

3.3.1.4 get_log_full_action

Returns the action take when the storage area becomes full.

Parameters and Return

Parameter	Type	Description
<return>	**LogFullAction**	The actually selected alternative of the LogFullAction enumeration.

Exceptions

This operation raises no exceptions.

Description

Since the storage space of the Log storage area is finite, the Logging Service has to take special action when the free space is depleted. The kind of action is described by the **LogFullAction** type. The **get_log_full_action** operation returns the information about which action the Logging Service will take when the storage area becomes full. The possible values are **HALT**, which means no further logging records are accepted and stored; or **WRAP**, which means the **Log** continues by overwriting the oldest records in the storage area.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **getLogFullAction**, defined in the UML class Log and made visible through interface **LogStatus** in the PIM (See Section 2.3.4, "getLogFullAction," on page 2-10).

The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name and signature the equivalent operation of the Telecom Log Service; however, the result is different.

3.3.1.5 get_availability_status

Returns the availability status of the Log.

Parameters and Return

Parameter	Type	Description
`<return>`	`AvailabilityStatus`	An instance of the AvailabilityStatus representing the actual status of the log

Exceptions

This operation raises no exceptions.

Description

The ability of the Log to accept and store logging records might become impaired. The `get_availability_status` operation is used to check the availability status of the Log. The returned instance of the **AvailibilityStatus** type contains two Boolean values: **off_duty**, which indicates the log is disabled when true; and **log_full**, which indicates that all free space is depleted in the log storage area.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **getAvailabilityStatus**, defined in the UML class **Log** and made visible through interface **LogStatus** in the PIM (See Section 2.3.5, "getAvailabilityStatus," on page 2-11). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name, signature and result to the equivalent operation of the Telecom Log Service.

3.3.1.6 get_administrative_state

Returns the administrative state of the Log.

Parameters and Return

Parameter	Type	Description
`<return>`	`AdministrativeState`	The actually selected alternative of the AdministrativeState enumeration.

Exceptions

This operation raises no exceptions.

Description

The ability of the logging service to accept and store new logging records can be affected by administrative action. The `get_administrative_state` is used to read the administrative state of the Log. The possible states are **locked** and **unlocked**. If the state is **locked**, no new records are accepted. Reading of already stored records is not affected.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **getAdministrativeState**, defined in the UML class **Log** and made visible through interface **LogStatus** in the PIM (See Section 2.3.6, "getAdministrativeState," on page 2-11). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name, signature and result to the equivalent operation of the Telecom Log Service.

3.3.1.7 *get_operational_state*

Returns the operational state of the Log.

Parameters and Return

Parameter	Type	Description
`<return>`	`OperationalState`	The actually selected alternative of the OperationalState enumeration.

Exceptions

This operation raises no exceptions.

Description

The **`get_operational_state`** operation returns the actual operational state of the log. Possible values are **enabled**, which means the log is fully functional and available to log producer and log consumer clients; or **disabled**, which indicates the log has encountered a runtime problem and is not available for use by log producers or log consumers.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **getOperationalState**, defined in the UML class **Log** and made visible through interface **LogStatus** in the PIM (See Section 2.3.7, "getOperationalState," on page 2-12). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name, signature and result to the equivalent operation of the Telecom Log Service.

3.3.2 Interface LogConsumer

```
interface LogConsumer : LogStatus {
     RecordId get_record_id_from_time (in LogTime fromTime)
                          raises(InvalidParam);
     LogRecordSequence retrieve_by_id(inout RecordId currentId,
                                   inout unsigned long howMany)
                          raises(InvalidParam);
};
```

3.3.2.1 get_record_id_from_time

Identify a record in the log a record based on its time stamp.

Parameters and Return

Parameter	Type	Description
fromTime	**LogTime**	The timestamp to start the search with.
<return>	**RecordId**	Record ID of the first record matching the timestamp.

Exceptions

This operation raises the InvalidParam exception if the supplied parameter is invalid.

Description

The `get_record_id_from_time` operation returns the record Id of the first record in the Log with 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 the criteria provided, then the **RecordId** returned corresponds to the next record that will be recorded in the future. In this way, if this "future" `recordId` is passed into the `retrieveById` operation, an empty record will be returned unless records have been recorded since the time specified. Note that if the time specified in the `fromTime` parameter is in the future, there is no guarantee that the resulting records returned by the `retrieveById` operation will have a time stamp after the `fromTime` parameter if the returned `recordId` from this invocation of the `get_record_id_from_time` operation is subsequently used as input to the `retrieveById` operation.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **getRecordIdFromTime**, defined in the UML class **Log** and made visible through interface **LogConsumer** in the PIM (See Section 2.4.1, "getRecordIdFromTime," on page 2-12). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This is a new operation, not available in the Telecom Log Service. It reflects the architectural and operational difference between the two services.

3.3.2.2 *retrieve_by_id*

Retrieves a specified number of records from the Log.

Parameters and Return

Parameter	Type	Description
`currentId`	`RecordId`	The ID of the starting record.
`howMany`	`Unsigned long`	The number of records to retrieve.
`<return>`	`LogRecordSequence`	The sequence of retrieved records.

Exceptions

This operation raises the InvalidParam exception if the supplied parameter is invalid.

Description

The **retrieve_by_id** operation returns a **LogRecordSequence** that begins with the record specified by the **currentId** parameter. The number of records in the **LogRecordSequence** returned by the **retrieve_by_id** operation is equal to the number of records specified by the **howMany** parameter, or the number of records available if the number of records specified by the **howMany** parameter cannot be met. The log will update **howMany** to indicate the number of records returned and will set **currentId** to the id of the record following the last retrieved record. If there are no further records available, **currentId** will be set to zero. If the record specified by **currentId** does not exist, or if the Log is empty, the **retrieve_by_id** operation returns an empty list of **LogRecords**, and sets both, **currentId** and **howMany** to zero.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **retrieveById**, defined in the UML class **Log** and made visible through interface **LogConsumer** in the PIM (See Section 2.4.2, "retrieveById," on page 2-13). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This is a new operation, not available in the Telecom Log Service. It reflects the architectural and operational difference between the two services.

3.3.3 Interface LogProducer

```
interface LogProducer : LogStatus {
    typedef sequence<LogProducerRecord>
                        LogProducerRecordSequence;
  oneway void write_records(
                in ProducerLogRecordSequence records);
};
```

This interface allows the insertion of new log records into the logging storage area encapsulated by the Log class. In favor of preserving the overall operational integrity of the system, no guarantee is made that a logging record is accepted and stored if the logging service is unable to process and /or store it.

3.3.3.1 write_records

Writes records to the Log.

Parameters and Return

Parameter	Type	Description
records	**ProducerLogRecordSequence**	The records to be written to the log.
<return>	**void**	This operation provides no return.

Exceptions

This operation raises no exceptions.

Description

The **write_records** operation adds the log records supplied in the **records** parameter to the Log. When there is insufficient storage to add one of the supplied log records to the Log, and the **LogFullAction** is set to **HALT**, the **write_records** operation shall set the availability status logFull state to true. For example, if 3 records are provided in the records parameter, and while trying to write the second record to the log, the record will not fit, then the log is considered to be full. Therefore, the second and third records will not be stored in the log but the first record would have been successfully stored..

The **write_records** operation inserts the current UTC time to the **time** field of each record written to the Log, and assigns a unique record id to the **id** field of the **LogRecord**.

Log records accepted for storage by the **write_records** will be available for retrieval in the order received.

Note – The purpose of the oneway invocation is, within the limitations of embedded ORBs, to de-couple the log producer from the logging service implementation, so that difficulties in the Log have no side-effects on the log producer or its operation. However, since ORBs may legally discard oneway requests, implementers should take extra care that the onway invocations of **write_records** are not discarded without very substantial reason.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **writeRecords**, defined in the UML class **Log** and made visible through interface **LogProducer** in the PIM (See Section 2.5.1, "writeRecords," on page 2-14). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This is a new operation, not available in the Telecom Log Service. It reflects the
architectural and operational difference between the two services.

3.3.4 Interface LogAdministrator

```
interface LogAdministrator : LogStatus {
   void set_max_size(in unsigned long long size)
               raises (InvalidParam);
   void set_log_full_action(in LogFullAction action);
   void set_administrative_state(in AdministrativeState state);
   void clear_log();
   void destroy ();
};
```

This interface allows the retrieval of logging records from the storage area encapsulated
by the Log class.

3.3.4.1 set_max_size

Sets the maximum size the Log storage area.

Parameters and Return

Parameter	Type	Description
`size`	`unsigned long long`	The desired size for the logging storage area in bytes.
`<return>`	`void`	This operation does not return a value.

Exceptions

This operation raises the InvalidParam exception if the supplied parameter is invalid.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The available
space in this storage area is finite. This operation allows setting of the maximum capacity
in bytes of the storage area. Note, however, that this operation might be constraint by the
underlying operation (you can't assign more memory than is physically present), or a
platform specific implementation might decide to render this operation as a no-op and
provide a fixed maximum size instead.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **setMaxSize**, defined in the UML class **Log** and made visible through interface **LogController** in the PIM (See Section 2.6.1, "setMaxSize," on page 2-15). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name and signature to the equivalent operation of the Telecom Log Service; however, a result return has been added.

3.3.4.2 *set_log_full_action*

Configure the action to be taken if the log storage area becomes full.

Parameters and Return

Parameter	Type	Description
`action`	`LogFullAction`	Specify the desired selection from the LogFullAction enumeration (either HALT or WRAP).
`<return>`	`void`	

Exceptions

This operation raises no exceptions.

Description

Since the storage space of the Log storage area is finite, the Logging Service has to take special action when the free space is depleted. The kind of action is described by the **LogFullAction** type. The `set_log_full_action` operation allows the specification which action should be taken after all free space in the log storage area is depleted. The possible values are **HALT**, which means no further logging records are accepted and stored; or **WRAP**, which means the **Log** continues by overwriting the oldest records in the storage area.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **setLogFullAction**, defined in the UML class **Log** and made visible through interface **LogController** in the PIM (See Section 2.6.2, "setLogFullAction," on page 2-16). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is in principle identical in name, signature and return to the equivalent operation of the Telecom Log Service; however, the input parameter type has been changed to an IDL enumeration.

3.3.4.3 set_administrative_state

The `set_administrative_state` operation provides write access to the administrative state value.

Parameters and Return

Parameter	Type	Description
`state`	`unsigned long long`	Select the desired alternative from the AdministrativeState enumeration. (Possible values are **locked** and **unlocked**).
`<return>`	`void`	

Exceptions

This operation raises no exceptions.

Description

This operation allows one to affect the ability of the logging service to accept and store new logging records by administrative action. The possible states are **LOCKED** and **UNLOCKED**. If the state is **LOCKED**, no new records are accepted. Reading of already stored records is not affected. If the state is set to **UNLOCKED**, the log operates normally.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **setLogFullAction**, defined in the UML class **Log** and made visible through interface **LogController** in the PIM (See Section 2.6.2, "setLogFullAction," on page 2-16). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name, signature, and result to the equivalent operation of the Telecom Log Service.

3.3.4.4 clear_log

Purge the log storage area.

Parameters and Return

This operation has no parameters or returns.

Exceptions

This operation raises no exceptions.

Description

This operation purges all logging records from the log storage area; however, it does not alter the size of the storage area in any way. A subsequent invocation of the **get_current_size** operation will return 0 (zero).

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **setLogFullAction**, defined in the UML class **Log** and made visible through interface **LogController** in the PIM (See Section 2.6.2, "setLogFullAction," on page 2-16). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name, signature and result to the equivalent operation of the Telecom Log Service.

3.3.4.5 destroy

Tear down an instantiated Log.

Parameters and Return

This operation has no parameters or returns.

Exceptions

This operation raises no exceptions.

Description

This operation will destroy the associated instance of the **Log** class. All existing records in the log storage area are irrecoverably lost and the memory resources associated with the storage area are released.

Mapping from the Platform Independent Model

This IDL operation definition is the result of a one-to-one mapping from operation **setLogFullAction**, defined in the UML class **Log** and made visible through interface **LogController** in the PIM (See Section 2.6.2, "setLogFullAction," on page 2-16). The spelling of the operation name has been changed to conform to the OMG IDL Style Guide.

Difference to the Telecom Log Service

This operation is identical in name, signature and result to the equivalent operation of the Telecom Log Service.

3

Complete Logging Service IDL   *4*

#ifndef MODULE_COS_LW_LOG_IDL
#define MODULE_COS_LW_LOG_IDL

#PRAGMA prefix "omg.org"

```
module CosLwLog {
    // The following constants are intended to identify
    // the nature of a logging record. The constants
    // represent the valid values for LogLevel
    // The list of constants may be expanded
    const unsigned short SECURITY_ALARM = 1;
    const unsigned short FAILURE_ALARM = 2;
    const unsigned short DEGRADED_ALARM =3;
    const unsigned short EXCEPTION_ERROR =4;
    const unsigned short FLOW_CONTROL_ERROR =5;
    const unsigned short RANGE_ERROR =6;
    const unsigned short USAGE_ERROR = 7;
    const unsigned short ADMINISTRATIVE_EVENT = 8;
    const unsigned short STATISTIC_REPORT = 9;
    // Values ranging from 10 to 26 are reserved for
    // 16 debugging levels.

    typedef unsigned short LogLevel;
    enum OperationalState {disabled, enabled};
    enum AdministrativeState {locked, unlocked};
    enum LogFullAction (WRAP, HALT);
    typedef unsigned long long RecordId;
    struct LogTime {
    long seconds;
    long nanoseconds;
    };
```

```
struct AvailabilityStatus{
    boolean off_duty;
    boolean log_full;
};
struct ProducerLogRecord {
    string   producerId;
    string   producerName;
    LogLevel  level;
    string   logData;
};
struct LogRecord {
    RecordId       id;
    LogTime        time;
    ProducerLogRecord info;
};
typedef sequence<LogRecord> LogRecordSequence;

interface LogStatus {
    unsigned long long get_max_size();
    unsigned long long get_current_size();
    unsigned long long get_n_records();
    LogFullAction get_log_full_action();
    AvailabilityStatus get_availability_status();
    AdministrativeState get_administrative_state();
    OperationalState get_operational_state();
};

interface LogConsumer : LogStatus {
    RecordId get_record_id_from_time (in LogTime fromTime)
        raises(InvalidParam);
    LogRecordSequence retrieve_by_id(
            inout RecordId    currentId,
            inout unsigned long howMany)
        raises(InvalidParam);
};

interface LogProducer : LogStatus {
typedef sequence<LogProducerRecord>
    LogProducerRecordSequence;
oneway void write_records(
            in ProducerLogRecordSequence records);
};

interface LogAdministrator : LogStatus {
    void set_max_size(in unsigned long long size)
        raises (InvalidParam);
    void set_log_full_action(in LogFullActionType action);
    void set_administrative_state(
            in AdministrativeStateType state);
    void clear_log();
    void destroy ();
```

```
        };

};
#endif   // MODULE_COS_LW_LOG_IDL
```

4