The Consultative Committee for Space Data Systems

Recommendation for Space Data System Practices

SPACE LINK EXTENSION—APPLICATION PROGRAM INTERFACE FOR TRANSFER SERVICES—CORE SPECIFICATION

RECOMMENDED PRACTICE

CCSDS 914.0-M-1

Magenta Book

October 2008

The Consultative Committee for Space Data Systems

Recommendation for Space Data System Practices

SPACE LINK EXTENSION—APPLICATION PROGRAM INTERFACE FOR TRANSFER SERVICES—CORE SPECIFICATION

RECOMMENDED PRACTICE

CCSDS 914.0-M-1

Magenta Book

October 2008

AUTHORITY

Issue:	Recommended Practice, Issue 1
Date:	October 2008
Location:	Washington, DC, USA

This document has been approved for publication by the Management Council of the Consultative Committee for Space Data Systems (CCSDS) and represents the consensus technical agreement of the participating CCSDS Member Agencies. The procedure for review and authorization of CCSDS documents is detailed in the *Procedures Manual for the Consultative Committee for Space Data Systems*, and the record of Agency participation in the authorization of this document can be obtained from the CCSDS Secretariat at the address below.

This document is published and maintained by:

CCSDS Secretariat
Space Communications and Navigation Office, 7L70
Space Operations Mission Directorate
NASA Headquarters
Washington, DC 20546-0001, USA

STATEMENT OF INTENT

The Consultative Committee for Space Data Systems (CCSDS) is an organization officially established by the management of its members. The Committee meets periodically to address data systems problems that are common to all participants, and to formulate sound technical solutions to these problems. Inasmuch as participation in the CCSDS is completely voluntary, the results of Committee actions are termed **Recommendations** and are not in themselves considered binding on any Agency.

CCSDS Recommendations take two forms: **Recommended Standards** that are prescriptive and are the formal vehicles by which CCSDS Agencies create the standards that specify how elements of their space mission support infrastructure shall operate and interoperate with others; and **Recommended Practices** that are more descriptive in nature and are intended to provide general guidance about how to approach a particular problem associated with space mission support. This **Recommended Practice** is issued by, and represents the consensus of, the CCSDS members. Endorsement of this **Recommended Practice** is entirely voluntary and does not imply a commitment by any Agency or organization to implement its recommendations in a prescriptive sense.

No later than five years from its date of issuance, this **Recommended Practice** will be reviewed by the CCSDS to determine whether it should: (1) remain in effect without change; (2) be changed to reflect the impact of new technologies, new requirements, or new directions; or (3) be retired or canceled.

In those instances when a new version of a **Recommended Practice** is issued, existing CCSDS-related member Practices and implementations are not negated or deemed to be non-CCSDS compatible. It is the responsibility of each member to determine when such Practices or implementations are to be modified. Each member is, however, strongly encouraged to direct planning for its new Practices and implementations towards the later version of the Recommended Practice.

FOREWORD

Through the process of normal evolution, it is expected that expansion, deletion, or modification of this document may occur. This Recommended Practice is therefore subject to CCSDS document management and change control procedures, which are defined in the *Procedures Manual for the Consultative Committee for Space Data Systems*. Current versions of CCSDS documents are maintained at the CCSDS Web site:

http://www.ccsds.org/

Questions relating to the contents or status of this document should be addressed to the CCSDS Secretariat at the address indicated on page i.

At time of publication, the active Member and Observer Agencies of the CCSDS were:

Member Agencies

- Agenzia Spaziale Italiana (ASI)/Italy.
- British National Space Centre (BNSC)/United Kingdom.
- Canadian Space Agency (CSA)/Canada.
- Centre National d'Etudes Spatiales (CNES)/France.
- China National Space Administration (CNSA)/People's Republic of China.
- Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)/Germany.
- European Space Agency (ESA)/Europe.
- Federal Space Agency (FSA)/Russian Federation.
- Instituto Nacional de Pesquisas Espaciais (INPE)/Brazil.
- Japan Aerospace Exploration Agency (JAXA)/Japan.
- National Aeronautics and Space Administration (NASA)/USA.

Observer Agencies

- Austrian Space Agency (ASA)/Austria.
- Belgian Federal Science Policy Office (BFSPO)/Belgium.
- Central Research Institute of Machine Building (TsNIIMash)/Russian Federation.
- Centro Tecnico Aeroespacial (CTA)/Brazil.
- Chinese Academy of Sciences (CAS)/China.
- Chinese Academy of Space Technology (CAST)/China.
- Commonwealth Scientific and Industrial Research Organization (CSIRO)/Australia.
- Danish National Space Center (DNSC)/Denmark.
- European Organization for the Exploitation of Meteorological Satellites (EUMETSAT)/Europe.
- European Telecommunications Satellite Organization (EUTELSAT)/Europe.
- Hellenic National Space Committee (HNSC)/Greece.
- Indian Space Research Organization (ISRO)/India.
- Institute of Space Research (IKI)/Russian Federation.
- KFKI Research Institute for Particle & Nuclear Physics (KFKI)/Hungary.
- Korea Aerospace Research Institute (KARI)/Korea.
- MIKOMTEK: CSIR (CSIR)/Republic of South Africa.
- Ministry of Communications (MOC)/Israel.
- National Institute of Information and Communications Technology (NICT)/Japan.
- National Oceanic and Atmospheric Administration (NOAA)/USA.
- National Space Organization (NSPO)/Chinese Taipei.
- Naval Center for Space Technology (NCST)/USA.
- Space and Upper Atmosphere Research Commission (SUPARCO)/Pakistan.
- Swedish Space Corporation (SSC)/Sweden.
- United States Geological Survey (USGS)/USA.

DOCUMENT CONTROL

Document	Title	Date	Status
CCSDS 914.0-M-1	Space Link Extension—Application Program Interface for Transfer Services—Core Specification, Recommended Practice, Issue 1	October 2008	Original issue
EC 1	Editorial Change 1	December 2008	Updates references to recent publications

CONTENTS

CONTENTS (continued)

Section Page

Figure

CONTENTS (continued)

1 INTRODUCTION

1.1 PURPOSE OF THIS RECOMMENDED PRACTICE

The purpose of this Recommended Practice is to define a C++ Application Program Interface (API) for CCSDS Space Link Extension (SLE) Transfer Services, which is independent of any specific technology used for communications between an SLE service user and an SLE service provider.

This API is intended for use by application programs implementing SLE services. It can be configured to support SLE service user applications or SLE service provider applications.

This API is also intended to simplify the implementation of gateways that are required to achieve interoperability between SLE service provider and SLE service user applications using different communications technologies.

Using this Application Program Interface Recommended Practice, API implementations (software packages) able to run on specific platforms can be developed. Once developed, such a package can be supplied to new users of SLE services for integration with their user or production facilities, thus minimizing their investment to buy into SLE support.

1.2 SCOPE

1.2.1 ITEMS COVERED BY THIS RECOMMENDED PRACTICE

This Recommended Practice defines the Application Program Interface in terms of:

a) the components that provide the services of the API;

b) the functionality provided by each of the components;

c) the interfaces provided by each of the components; and

d) the externally visible behavior associated with the interfaces exported by the components.

It does not specify:

a) individual implementations or products;

b) the internal design of the components; and

c) the technology used for communications.

This Recommended Practice defines those aspects of the Application Program Interface, which are common for all SLE service types or for a subset of the SLE service types, e.g., all return link services or all forward link services. It also defines a framework for specification of service type-specific elements of the API. Service-specific aspects of the API are defined

by supplemental Recommended Practice documents for SLE return link services (references [10], [11], and [12]) and SLE forward link services (references [13] and [14]).

This Recommended Practice for the Application Program Interface responds to the requirements imposed on such an API by the CCSDS SLE transfer service Recommended Standards that were available when this Recommended Practice was released.

1.2.2 CONFORMANCE TO CCSDS RECOMMENDED STANDARDS

This version of the SLE API Recommended Practice conforms to the CCSDS Recommended Standards for Space Link Extension Services, referenced in 1.7, with the exception of the following optional features:

a) The negotiation procedure for version numbers in the BIND operation is not supported. If the responder does not support the version number identified in the BIND Invocation, it responds with a BIND Return containing a negative result and the diagnostic 'version number not supported'. The responder does not propose an alternative version number.

b) Provider-initiated binding, specified by CCSDS Recommended Standards for return link services is not included in this Recommended Practice. The management parameters that specify the bind initiative are supported to simplify addition of this procedure in later versions.

1.3 APPLICABILITY

For the SLE transfer services Return All Frames (RAF), Return Channel Frames (RCF), and Forward CLTU, the API specified in this document supports two versions, namely:

a) version 1 as specified by references [C1], [C2], and[C3]; and

b) version 2 as specified by references [4], [5], and [7].

Support for version 1 of these services is included for backward compatibility purposes for a limited time and may not be continued in future versions of this specification. Support for version 1 of the RAF, RCF and CLTU services implies that SLE API implementations of this specification are able to interoperate with peer SLE systems that comply with the specification of the Transport Mapping Layer (TML) in 'Specification of a SLE API Proxy for TCP/IP and ASN.1', ESOC, SLES-SW-API-0002-TOS-GCI, Issue 1.1, February 2001.

Version dependent provisions within this Recommended Practice are marked as follows:

– [V1:] for provisions specific to version 1 of RAF, RCF, or CLTU; and

– [V2:] for provisions specific to version 2 of RAF, RCF, or CLTU.

1.4 RATIONALE

This Recommended Practice describes the services provided by a software package implementing the API to application software using the API. It specifies the mapping of the SLE Transfer Services specifications to specific functions and parameters of the SLE API. It also specifies the distribution of responsibility for specific functions between SLE API software and application software. The distribution of responsibility has been defined with due consideration for reusability of software packages implementing the SLE API.

The goal of this Recommended Practice is to create a guide for interoperability between

a) software packages implementing the SLE API; and

b) application software using the SLE API.

This interoperability guide also allows exchangeability of different products implementing the SLE API, as long as they adhere to the interface specification of this Recommended Practice.

1.5 DOCUMENT STRUCTURE

1.5.1 OVERVIEW

This Recommended Practice is organized in two parts and a set of annexes.

1.5.1.1 Part I—The Descriptive Part

The descriptive part presents the API Model in section 2 using the Unified Modeling Language (UML), see reference [J6].

1.5.1.2 Part II—The Prescriptive Part

The prescriptive part contains the specification of the API. In case of any discrepancies between the descriptive part and the prescriptive part, the specifications in the latter shall apply.

Section 3 contains detailed specifications of the API components and of the interfaces that must be provided by the application.

Section 4 defines the state tables that must be implemented by the API.

1.5.1.3 Annexes

Annex A contains the detailed declaration of the C++ interfaces, which are common for all SLE service types.

Annex B lists the result codes that are used by the API.

[V1:] For version 1 of the services RAF, RCF, and CLTU, annex C defines a standard ASCII representation for the service instance identifier and lists the attribute identifiers and abbreviations that are valid for the service instance identifier.

[V2:] For later versions of these services and all other services, these specifications are provided by the applicable CCSDS Recommended Standards.

Annex D describes the design patterns and conventions that shall be applied to API components. The specifications in this annex are also relevant for the application software using the API.

Annex E defines requirements for software products claiming conformance with this Recommended Practice.

Annex F describes the interaction of API components, showing several use cases.

Annex G provides cross-references between interfaces provided by API components and interfaces used by API components.

Annex H contains an index to definitions.

Annex I explains all acronyms used in this Recommended Practice.

Annex J lists informative reference documents.

1.5.2 DOCUMENTATION TREE FOR SLE SERVICES AND SLE API

This Recommended Practice is based on the cross support model defined in the SLE Reference Model (reference [3]). The SLE services constitute one of the three types of Cross Support Services:

a) Part 1: SLE Services;

b) Part 2: Ground Domain Services; and

c) Part 3: Ground Communications Services.

The SLE services are further divided into SLE Service Management and SLE Transfer Services.

NOTE – In reference [3], SLE transfer services are identified; however, the complete service specifications will be provided in separate Recommended Standards.

This Recommended Practice describes how the functions of an SLE transfer service provider or user can be implemented in a software package for the purpose of providing or using one or several SLE transfer services. It is part of a suite of documents specifying the API for SLE transfer services:

a) Core Specification of the Application Program Interface for Transfer Services (this Recommended Practice);

b) a set of Application Program Interfaces for specific Transfer Services; and

c) Internet Protocol for Transfer Services.

The basic organization of the SLE services and SLE API documentation is shown in figure 1-1. The various documents are described in the following paragraphs.

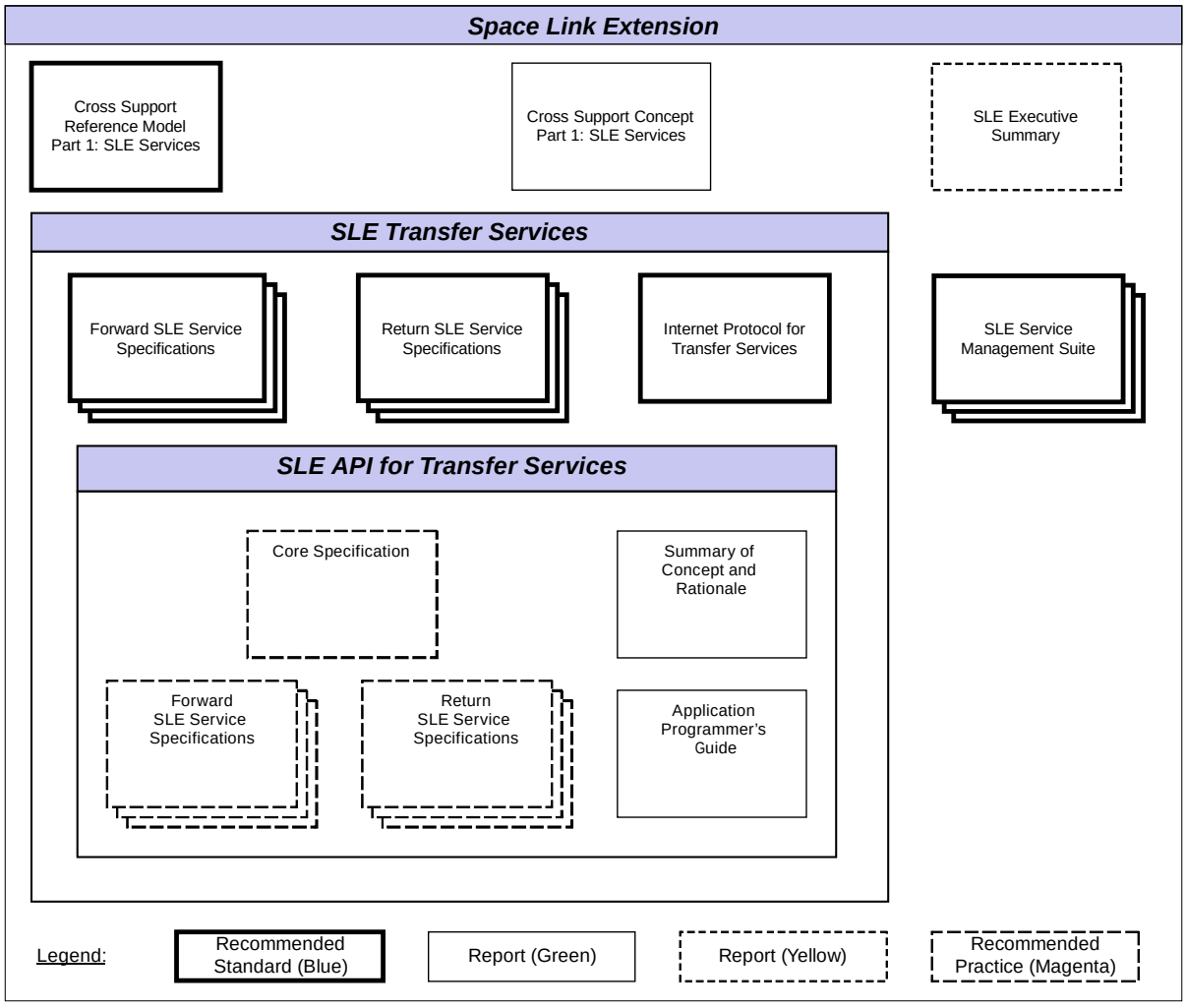

Figure 1-1: SLE Services and SLE API Documentation

a) *Cross Support Reference Model—Part 1: Space Link Extension Services*, a Recommended Standard that defines the framework and terminology for the specification of SLE services.

b) *Cross Support Concept—Part 1: Space Link Extension Services*, a Report introducing the concepts of cross support and the SLE services.

c) *Space Link Extension Services—Executive Summary*, an Administrative Report providing an overview of Space Link Extension (SLE) Services. It is designed to assist readers with their review of existing and future SLE documentation.

d) *Forward SLE Service Specifications*, a set of Recommended Standards that provide specifications of all forward link SLE services.

e) *Return SLE Service Specifications*, a set of Recommended Standards that provide specifications of all return link SLE services.

f) *Internet Protocol for Transfer Services*, a Recommended Standard providing the specification of the wire protocol used for SLE transfer services.

g) *SLE Service Management Specifications*, a set of Recommended Standards that establish the basis of SLE service management.

h) *Application Program Interface for Transfer Services—Core Specification*, this document.

i) *Application Program Interface for Transfer Services—Summary of Concept and Rationale*, a Report describing the concept and rationale for specification and implementation of a Application Program Interface for SLE Transfer Services.

j) *Application Program Interface for Return Services*, a set of Recommended Practice documents specifying the service-type specific extensions of the API for return link SLE services.

k) *Application Program Interface for Forward Services*, a set of Recommended Practice documents specifying the service-type specific extensions of the API for forward link SLE services.

l) *Application Program Interface for Transfer Services—Application Programmer's Guide*, a Report containing guidance material and software source code examples for software developers using the API.

1.6 DEFINITIONS

1.6.1 DEFINITION OF TERMS USED IN THIS DOCUMENT

1.6.1.1 Definitions from the SLE Reference Model

This Recommended Practice makes use of the following terms defined in reference [3]:

a) invoker;

b) offline delivery mode;

c) online delivery mode;

d) operation;

e) performer;

f) service provider (provider);

g) service user (user);

h) SLE protocol data unit (SLE-PDU);

i) SLE transfer service instance (service instance);

j) SLE transfer service production (service production);

k) SLE transfer service provision (service provision);

l) SLE transfer service provision period (provision period).

1.6.1.2 Definitions from the ISO Abstract Service Definitions and Conventions

This Recommended Practice makes use of the following terms defined in reference [19]:

a) initiator;

b) responder.

1.6.1.3 Definitions from SLE Transfer Service Specifications

This Recommended Practice makes use of the following terms defined in references [4], [5], [6], [7], and [8]:

a) association;

b) communications service;

c) confirmed operation;

d) invocation;

e) parameter (of an operation);

f) port identifier;

g) return;

h) unconfirmed operation.

1.6.1.4 Additional Definitions

1.6.1.4.1 General

For the purpose of this Recommended Practice, the following definitions also apply:

1.6.1.4.2 Component

A software module, providing a well-defined service via a set of interfaces. In this document the term component is used only to refer to the API components defined by this Recommended Practice.

1.6.1.4.3 Component Interface

An interface exported by a component.

1.6.1.4.4 Component Object

An object within a component that can be accessed by one or more interfaces exported by the component. Objects providing more than one interface support navigation between these interfaces.

1.6.1.4.5 Client

A software entity that uses the services of a component or of an object by invocation of the methods of an interface provided by the component or object. In this Recommended Practice the qualified term ‘local client’ is used to stress the difference between an interface to a software entity on the same computer and the interface between an SLE service user and an SLE service provider.

1.6.1.4.6 Interface

The abstraction of a service that only defines the operations supported by that service, but not their implementations. The specification of an operation is referred to as a method.

1.6.2 NOTES ON TERMINOLOGY

1.6.2.1 General

The following conventions apply throughout this Recommended Practice:

a) the words ‘shall’ and ‘must’ imply a binding and verifiable specification;

b) the word ‘should’ implies an optional, but desirable, specification;

c) the word ‘may’ implies an optional specification;

d) the words ‘is’, ‘are’, and ‘will’ imply statements of fact.

1.6.2.2 Use of the Term ‘Client’

In a complete SLE API, the API Proxy interacts with the API Service Element in the same process and with a peer proxy across the network. However, the proxy might also be used in an environment where some other software entity interfaces locally to the proxy and supplies the interfaces defined for the API Service Element. An example for such a configuration is a gateway. Therefore, this specification uses the term ‘client’ when referring to the entity with which the proxy interacts locally.

Where it is necessary to explicitly distinguish between interaction with the peer proxy across the network and interactions with the client on the local computer, the qualified term ‘local client’ is used.

1.6.2.3 Use of the Term ‘Release’

The term ‘release an object’ (or a resource) must be understood to mean that all actions shall be taken that are required to free the allocated memory or other operating system resources. For interfaces defined in this specification, this means that the method `Release()` must be called for every reference to an interface that a component holds. (See annex D for a description of the method `Release()` and of the reference counting scheme.)

1.7 REFERENCES

The following documents contain provisions which, through reference in this text, constitute provisions of this Recommended Practice. At the time of publication, the editions indicated were valid. All documents are subject to revision, and users of this Recommended Practice are encouraged to investigate the possibility of applying the most recent editions of the documents indicated below. The CCSDS Secretariat maintains a register of currently valid CCSDS documents.

NOTE – A list of informative references is provided in annex J.

[1] *Time Code Formats*. Recommendation for Space Data System Standards, CCSDS 301.0-B-3. Blue Book. Issue 3. Washington, D.C.: CCSDS, January 2002.

[2] *Cross Support Concept — Part 1: Space Link Extension Services*. Report Concerning Space Data System Standards, CCSDS 910.3-G-3. Green Book. Issue 3. Washington, D.C.: CCSDS, March 2006.

[3] *Cross Support Reference Model—Part 1: Space Link Extension Services*. Recommendation for Space Data System Standards, CCSDS 910.4-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, October 2005.

[4] *Space Link Extension—Return All Frames Service Specification*. Recommendation for Space Data System Standards, CCSDS 911.1-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, December 2004.

[5] *Space Link Extension—Return Channel Frames Service Specification*. Recommendation for Space Data System Standards, CCSDS 911.2-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, December 2004.

[6] *Space Link Extension—Return Operational Control Fields Service Specification*. Recommendation for Space Data System Standards, CCSDS 911.5-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, December 2004.

[7] *Space Link Extension—Forward CLTU Service Specification*. Recommendation for Space Data System Standards, CCSDS 912.1-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, December 2004.

[8] *Space Link Extension—Forward Space Packet Service Specification*. Recommendation for Space Data System Standards, CCSDS 912.3-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, December 2004.

[9] *Space Link Extension—Internet Protocol for Transfer Services*. Recommendation for Space Data System Standards, CCSDS 913.1-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, September 2008.

[10] *Space Link Extension—Application Program Interface for Return All Frames Service*. Specification Concerning Space Data System Standards, CCSDS 915.1-M-1. Magenta Book. Issue 1. Washington, D.C.: CCSDS, October 2008.

[11] *Space Link Extension—Application Program Interface for Return Channel Frames Service*. Specification Concerning Space Data System Standards, CCSDS 915.2-M-1. Magenta Book. Issue 1. Washington, D.C.: CCSDS, October 2008.

[12] *Space Link Extension—Application Program Interface for Return Operational Control Fields Service*. Specification Concerning Space Data System Standards, CCSDS 915.5-M-1. Magenta Book. Issue 1. Washington, D.C.: CCSDS, October 2008.

[13] *Space Link Extension—Application Program Interface for the Forward CLTU Service*. Specification Concerning Space Data System Standards, CCSDS 916.1-M-1. Magenta Book. Issue 1. Washington, D.C.: CCSDS, October 2008.

[14] *Space Link Extension—Application Program Interface for the Forward Space Packet Service*. Specification Concerning Space Data System Standards, CCSDS 916.3-M-1. Magenta Book. Issue 1. Washington, D.C.: CCSDS, October 2008.

[15] *Information Technology—Abstract Syntax Notation One (ASN.1): Specification of Basic Notation*. International Standard, ISO/IEC 8824-1:2002. 3rd ed. Geneva: ISO, 2002.

[16] *Information Technology—ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. International Standard, ISO/IEC 8825-1:2002. 3rd ed. Geneva: ISO, 2002.

[17] *Information Technology—Open Systems Interconnection—The Directory: Models*. International Standard, ISO/IEC 9594-2:1998. 3rd ed. Geneva: ISO, 1998.

[18] *Information Technology—Open Systems Interconnection—The Directory: Public-Key and Attribute Certificate Frameworks*. International Standard, ISO/IEC 9594-8:2001. 4th ed. Geneva: ISO, 2001.

[19] *Information Technology—Text Communication—Message-Oriented Text Interchange Systems (MOTIS)—Part 3: Abstract Service Definition Conventions*. International Standard, ISO/IEC 10021-3:1990 [Withdrawn]. Geneva: ISO, 1990.

[20] *Programming Languages—C++*. International Standard, ISO/IEC 14882:2003. 2nd ed. Geneva: ISO, 2003.

[21] *Secure Hash Standard*. Federal Information Processing Standards Publication 180-1. Gaithersburg, MD: NIST, 1995.

2 DESCRIPTION OF THE SLE API

2.1 INTRODUCTION

2.1.1 SCOPE OF THE MODEL

The intention of this section is to provide a high-level yet precise description of the API covering all API components and their interaction. For this purpose, the section uses an object model presented in the Unified Modeling Language (UML). Detailed specifications for each of the components are provided in section 3, which references the concepts, objects, and interfaces described by this model.

Note that the material presented here is an API design, to the extent that the API is broken down into components and the interfaces and interactions of these components are specified. However, this model (i.e., design) is restricted to what must be defined to ensure co-operation between components and excludes specification of the internal design of components.

The model defines:

a) the runtime components, from which the API is constructed;

b) the externally visible logical architecture of the API in terms of:

   1) the interfaces that are exposed by the components;

   2) the functionality to which these interfaces provide access; and

   3) the behavior of the operations defined by the interfaces.

In order to specify the externally visible architecture, the model defines logical entities below the level of runtime components. These entities are to be understood as abstract analysis objects. It is not the intention to prescribe the structure defined by these objects for an implementation in any way. The only requirement for an implementation is to provide the interfaces specified with the functionality and the behavior described by the analysis objects.

Some minor semantic extensions to UML have been defined to highlight the difference between those aspects of the model that must be implemented as specified and those aspects that are required only for a complete and unambiguous description. Subsection 2.2 provides details of how UML is used in this model.

This section contains only a summary description of interfaces. A complete specification of the methods and types is provided in annex A for all interfaces that are not service type specific. Service type-specific interfaces are detailed in supplemental Recommended Practice documents defining service-specific APIs.

2.1.2 PRESENTATION OF THE MODEL

The API model is presented as follows:

Subsection 2.2 describes how UML is used for this model. It does not provide an introduction to UML. For a description of UML, the reader is asked to refer to the UML specification (see reference [J6]), or to one of the textbooks on the subject (see references [J7] and [J8]).

Subsection 2.3 describes the 'logical view'. It contains a subsection for each of the API components:

a) API Proxy (see 2.3.2);

b) API Service Element (see 2.3.3);

c) SLE Operations (see 2.3.6); and

d) SLE Utilities (see 2.3.7).

Subsection 2.3.4 describes interfaces that must be implemented by more than one component and describes the application interface to the API. The logical view is complemented by annex F providing an overview of how the components interact.

2.2 SPECIFICATION METHOD AND NOTATION

2.2.1 INTRODUCTION

The architectural model for the SLE API is defined using the Unified Modeling Language (UML) as defined in reference [J6]. This subsection describes some specific aspects of how UML is used in this Recommended Practice.

A component in UML models a runtime object, e.g., an executable file, a dynamically linked library, or similar operating system objects. Therefore, the relationships that can be defined for a component in UML are limited:

a) a component can implement ('realize') and export an interface;

b) a component can depend on another component (more precisely on the interface exported by another component).

In this Recommended Practice, the UML component is used to refer to a component that:

a) is delivered as one or more linkable libraries;

b) is instantiated by a global 'creator function' defined in annex D;

c) is substitutable by a different component providing the same interfaces.

This specification requires these characteristics only for the top-level components API Proxy, API Service Element, SLE Operations, and SLE Utilities. These components are considerably complex and provide a rather large number of interfaces. In order to specify these interfaces, additional model constructs are needed. Following the general UML approach, this Recommended Practice uses UML classes with specific stereotypes to define special model objects. The specialized model objects are:

a) Interface;

b) Component Class (CoClass);

c) Component Internal Class; and

d) Entity.

They are shown in figure 2-1 together with some important relationships addressed later in this section. In addition, the model uses the UML utility class to represent functions that are not bound to any specific class.

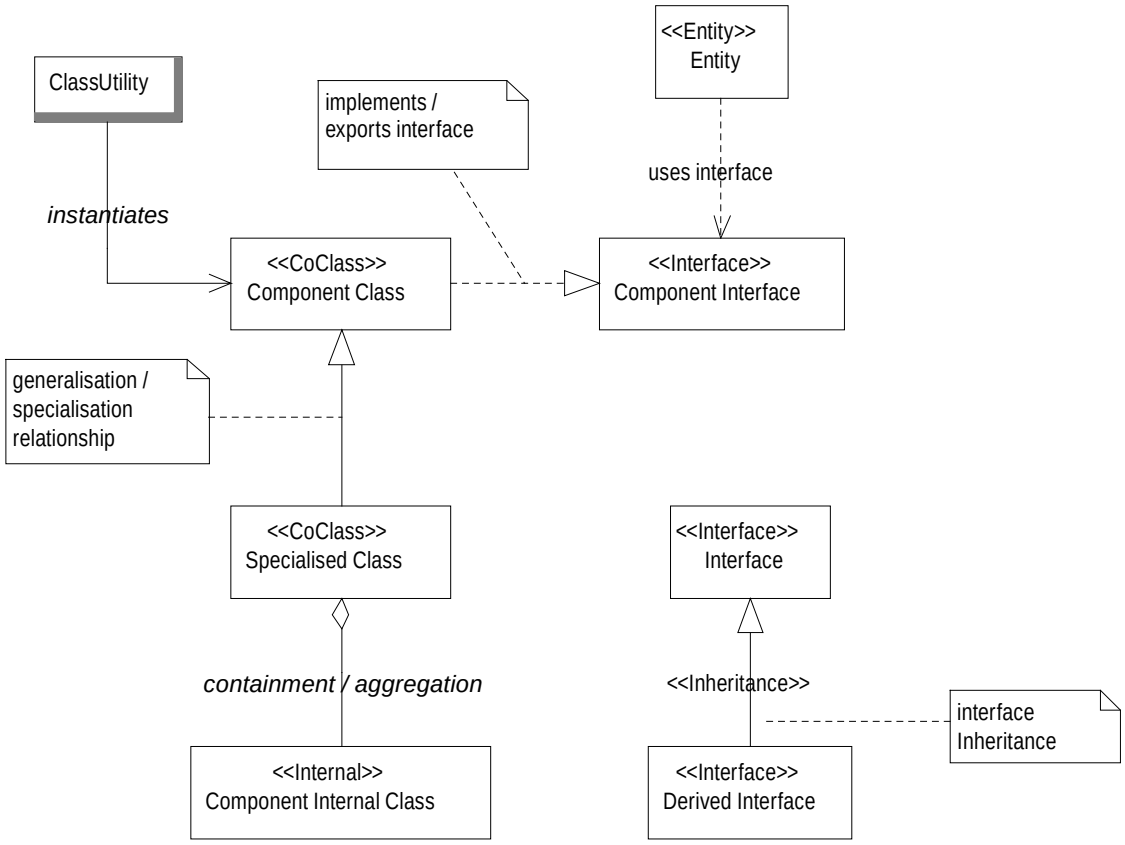

Figure 2-1: UML Stereotypes Used in This Recommended Practice

2.2.2 INTERFACE

The stereotype ‘Interface’ is defined by the UML specification. In this model it is used to identify a component interface. In C++ an interface is implemented as a class containing no data members and only public, pure virtual function members. According to the simple component model defined in annex D, all interfaces inherit the interface `IUnknown`. This fact is not explicitly shown in the diagrams.

An interface is displayed as a UML class with the stereotype <<Interface>>. The operations defined by the interface may or may not be shown, depending on the purpose of the specific diagram.

Where explicit public interface inheritance is required, this is indicated by the stereotype <<Inheritance>>. Generalization relationships that do not show this stereotype do not require an implementation using inheritance.

2.2.3 COMPONENT CLASS

A component class is a model object that specifies some functionality to be provided by a component. It is also used to describe navigational relationships between interfaces. The only implementation requirements related to component classes are the following. (For a description of the interface `IUnknown` and the method `QueryInterface()` see the ‘Simple Component Model’ in annex D.)

a) A component must export all interfaces specified for a component class it implements.

b) It must be possible to navigate between all interfaces specified for a component class and for component classes to which a generalization interface exists using `QueryInterface()`.

c) For every non-abstract component class (except the ‘main’ class for a component) the model defines one (or more) interfaces by which a new instance can be obtained. These interfaces must be supported.

d) When more than one instance of a component class exist, distinct references for the associated interfaces must be provided. The general requirement of the component model, that a query for the interface `IUnknown` on the same instance always returns the same pointer, applies.

Beyond these requirements, this Recommended Practice does not prescribe how the functionality defined for component classes is implemented. In particular, the generalization relationships shown in the model do not require implementation via inheritance. In fact, there need not be any equivalence between the classes within a component and the component classes shown in this model.

A component class is defined as abstract, when no instances of the class are created. Such component classes define common functionality, behavior, and interfaces that are provided by more than one derived class.

A runtime component contains a single 'main' class and exposes a special 'creator function' that can create an instance of that class. This creator function must be a global symbol in the library that implements the component. In the diagrams of this model the creator function is represented by a UML Utility Class, which has an association 'instantiates' to the 'main' class.

The model uses the UML dependency (or 'uses') relationship between component classes and interfaces to describe how components are linked via their interfaces. The only requirement for an implementation is that the component implementing the functionality associated with the component class use the specified interfaces for the purpose identified in the model.

In a few cases, attributes are shown for component classes. Attributes are strictly analysis-model constructs to highlight characteristics of a class or options provided by a class. They are not to be understood to define data. Attributes shown in the model may not even be accessible at all.

A component class is displayed as a UML class with the stereotype <<CoClass>>. If the component class is abstract, its name is displayed in *italic typeface.*

2.2.4 COMPONENT INTERNAL CLASS

Component internal classes are used to describe features that are expected from a component, but which do not result in any externally visible interface. Component internal classes are pure model objects. This specification does not prescribe how the features presented by these objects are implemented.

A component internal class is presented as a UML class with the stereotype <<Internal>>. An internal class does not implement an interface. Beside this constraint, all relationships for classes can be used.

2.2.5 ENTITY

In some cases, it is necessary to identify use of an interface by some entity, which is otherwise unspecified. For this purpose, the model object 'Entity' is used. An entity is displayed as a UML class with the stereotype <<Entity>>. The only relationship an entity can have is a dependency relationship to an interface. No further semantics are associated with an entity.

2.2.6 NAMING CONVENTIONS

2.2.6.1 Component Classes

Because component classes are not expected to be visible in source code, their names do not adhere to the syntax of identifiers in programming languages.

The names of component classes that are independent of service types are not specifically prefixed. Names of classes for which a special version must be provided for every service type are prefixed with '<SRV>'.

2.2.6.2 Interfaces

Interfaces adhere to the syntax of C/C++ identifiers. Except for diagrams, interface names and method names are displayed in `mono-space font`.

Following Component Object Model (COM) conventions (see reference [J5]), the name of an interface always starts with a capital 'I'. Interfaces that are independent of specific SLE service types are prefixed with `ISLE_`. Names of interfaces, which are specific for service types, are prefixed with `I<SRV>`. These interfaces are defined in supplemental Recommended Practice documents for service-specific APIs, where `<SRV>` is replaced by the abbreviation for the service type. Readability of the name following the prefix can be improved using upper and lower case letters. The underscore character is reserved for separation of prefixes from the name. It is not used in the name itself.

Examples: `ISLE_ProxyAdmin`
`ISLE_ServiceInform`
`I<SRV>_SIAdmin` becomes, e.g., `IFSP_SIAdmin` or `IRAF_SIAdmin`

2.2.6.3 Entities and Component Internal Classes

Because the objects are pure modeling constructs and are not expected to be visible in source code, their names do not adhere to the syntax of identifiers in programming languages and no special naming conventions are applied.

2.2.7 DYNAMIC MODELING

The API requires implementation of a number of state machines. Because these comprise a considerable number of states and events, this Recommended Practice uses state tables instead of the state diagrams foreseen by UML. Because implementation of these state tables is mandatory, they have been placed in the prescriptive part of this Recommended Practice.

2.3 LOGICAL VIEW

2.3.1 STRUCTURE

The logical view of the API is partitioned into the packages shown in figure 2-2, which also displays the dependencies between these packages. Dependencies on the package SLE Utilities are not shown in order to avoid overloading of the diagram.

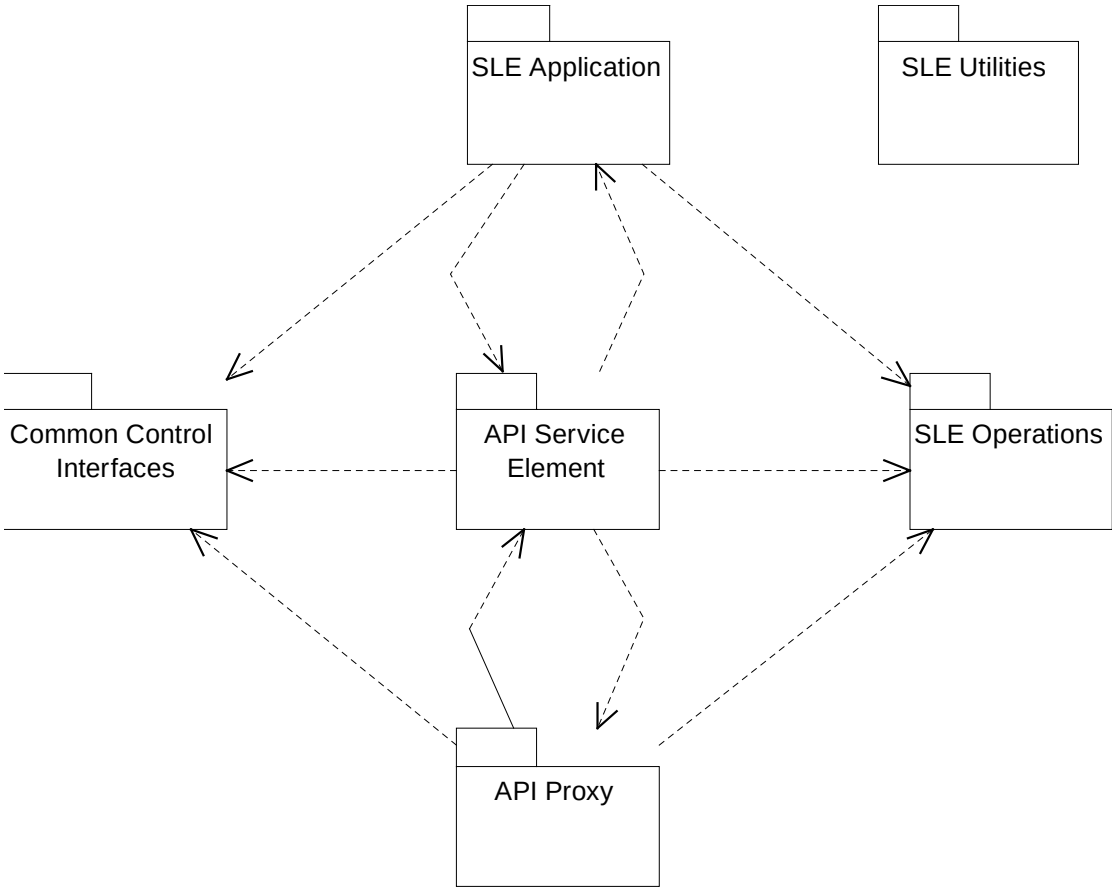

Figure 2-2: Top Level Decomposition of the API

The following packages exist:

a) API Proxy
The package contains the component classes that define the component API Proxy as well as interfaces exported only by the Proxy.

b) API Service Element
The package contains the component classes that define the component API Service Element as well as interfaces exported only by the API Service Element.

c) Common Control Interfaces
The package specifies some interfaces that are supported by the API Proxy and the API Service Element.

d) SLE Application
The package defines component classes that are assumed to be part of the application software. These component classes export the interfaces that must be made available by the application.

e) SLE Operations
The package specifies the interfaces for operation objects implemented by the associated component and used for SLE transfer service interfaces.

f) SLE Utilities
The package defines a small set of generally useful classes and their interfaces. The interfaces of these utility classes are used by interfaces throughout the model.

2.3.2 PACKAGE API PROXY

2.3.2.1 Overview

The API Proxy provides all functionality that must be implemented in a technology-specific manner and shields its clients from all technology-specific aspects. In addition, the Proxy implements access control on system level and authentication of the peer identity. Its structure is shown in figure 2-3.

The component class API Proxy is responsible for configuration, initialization, and management of the Proxy component and the data communication system. The configuration and initialization is performed using the interface `ISLE_ProxyAdmin`.

Communication between an SLE service user and an SLE service provider is handled by the class Association via the exported interface `ISLE_SrvProxyInitiate` and the complementary interface `ISLE_SrvProxyInform` supplied by the client. Associations can be created via the interface `ISLE_AssocFactory`.

Associations are distinguished according to the role they play in the BIND and UNBIND operation. Initiating associations invoke BIND and UNBIND operations, whereas responding associations accept incoming BIND and UNBIND invocations. These specialized classes differ in their behavior but do not expose any interfaces in addition to those inherited from the abstract class Association.

The PDU Translator is responsible for translation of the operation parameters between the syntax defined for the API and the syntax and encoding used for transfer. The abstract class PDU Translator handles common PDUs for association management, while a service-specific translator handles service-specific PDUs. It is the only element in the proxy that depends on the SLE service type.

The proxy and associations support logging and diagnostic traces using the interfaces `ISLE_Reporter` and `ISLE_Trace` provided by the application. Diagnostic traces can be switched on and off via the interface `ISLE_TraceControl` exported by the API Proxy and by the class Association.

NOTE – All classes in the package API Proxy use the interfaces of operation objects and of utility objects. This fact is not specifically mentioned in the following description.

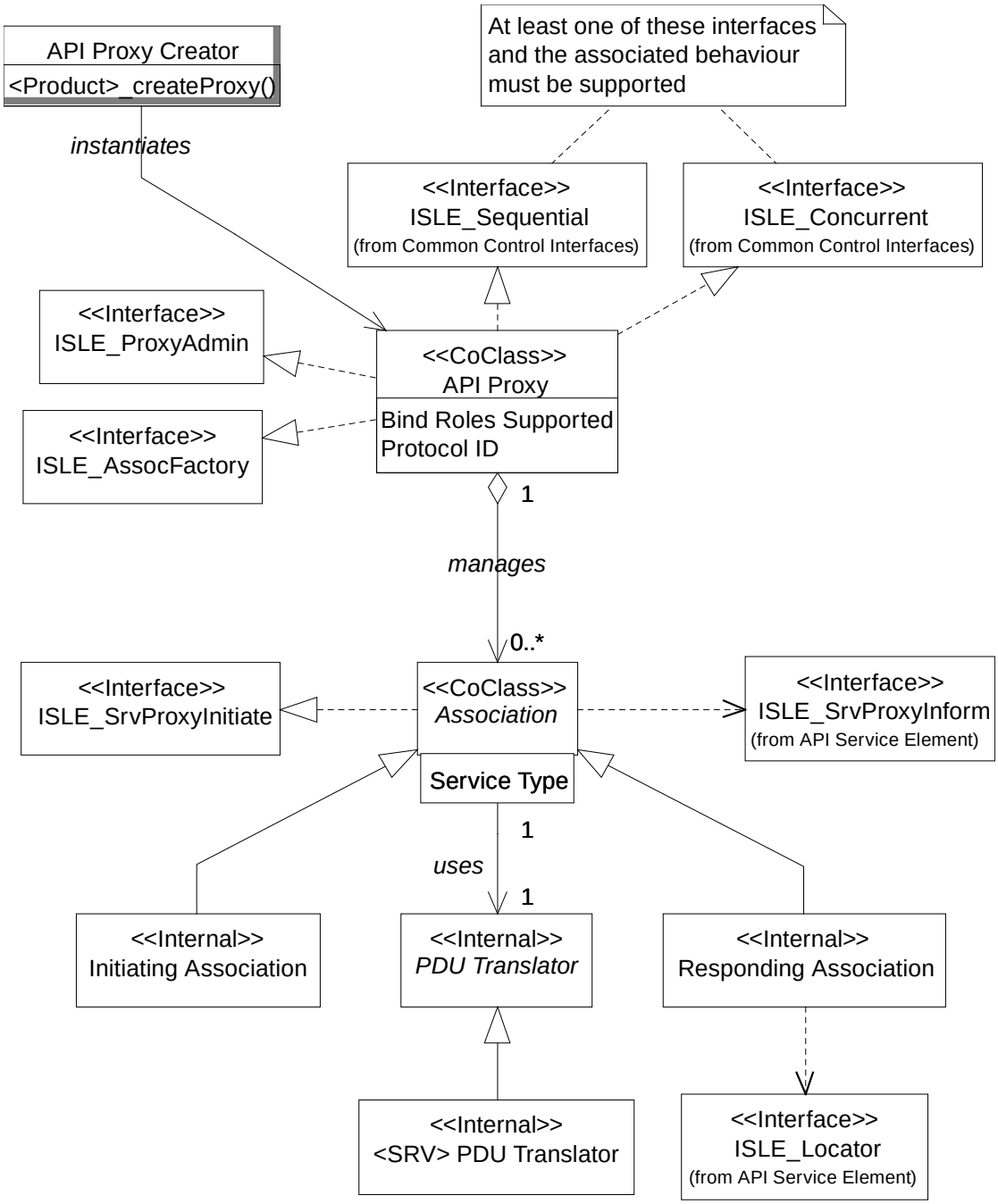

Figure 2-3: Structure of the Package API Proxy

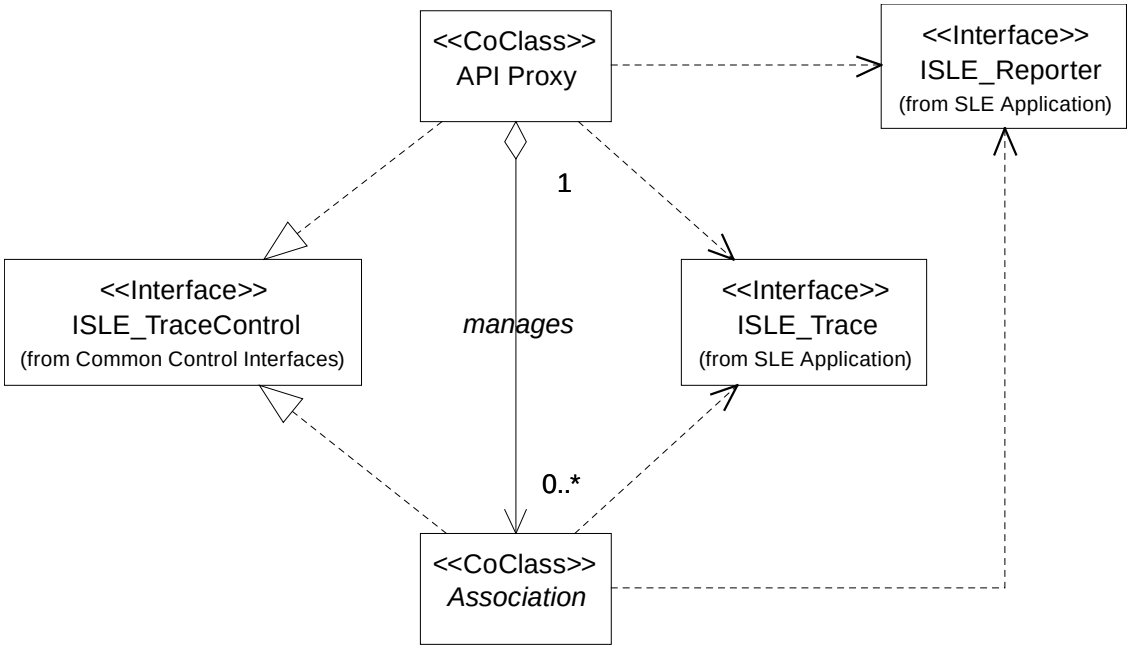

Figure 2-4: Reporting and Tracing by the Proxy

2.3.2.2 Component Class API Proxy

2.3.2.2.1 General

The API Proxy provides management of the communications infrastructure and of associations. The communications technology and the specific mapping of SLE transfer services to that technology by a proxy implementation is identified by a 'Protocol ID', available via the interface `ISLE_ProxyAdmin`.

In general, a proxy supports initiating associations and responding associations. However, an implementation may support only one of these roles. The 'bind roles' actually supported are defined by the attribute 'Bind Roles Supported'. A proxy supporting associations in the responder role listens for incoming connection requests on the network interface.

A single instance of this class exists within one instance of the component API Proxy.

2.3.2.2.2 Responsibilities

2.3.2.2.2.1 Configuration and Initialization of the Proxy Component

After creation the proxy must be configured and initialized using the interface `ISLE_ProxyAdmin`. This action includes configuration and initialization of the communications infrastructure. All static configuration parameters needed for this purpose are specified in the configuration database, defined in 2.3.2.8.

An implementation may require that part of the required infrastructure has already been initialized, e.g., at system start-up, or that a global infrastructure exists, e.g., for access to a directory system. Such prerequisites must be documented for every implementation.

2.3.2.2.2.2 Dynamic Port Registration and De-registration

The proxy performs dynamic registration of responder ports on request of a client via the interface `ISLE_ProxyAdmin`. Port registration includes all actions that may be required by a technology to register addresses, export address related information to a directory system, or publish a port by other means. Port registration has the effect that requests sent to the port are correctly routed to the proxy that registered it.

If port registration is not needed for the technology used, or for the current configuration of the proxy, the proxy ignores the request.

2.3.2.2.2.3 Management of Initiating Associations

The proxy creates and initializes initiating associations for a specified service type on request of its client via the interface `ISLE_AssocFactory`. If the proxy does not support the requested service type or does not support associations in the initiator role, it rejects the request.

The proxy keeps a reference to the associations until the client requests it to destroy the association. If the association is not in an unbound state, the proxy rejects this request. Otherwise, it releases all resources that may be allocated to the association and performs all actions required to delete the association.

2.3.2.2.2.4 Management of Responding Associations

A proxy that supports associations in the role of a BIND responder starts listening for incoming BIND invocations as soon as the start method of one of the control interfaces has been called, or when the port has been registered. When the proxy receives a BIND invocation, it creates a new responding association to process the BIND invocation.

When responding association terminates (following UNBIND, PEER-ABORT, or after a failure reported by the data communication service), the proxy releases all resources allocated to the association and deletes the association object.

2.3.2.2.2.5 Logging and Notification

The proxy generates log records for important events and enters them to the system log using the interface `ISLE_Reporter` provided by the application. For specific events that require immediate attention, the proxy notifies the application using the method `Notify()` in the interface `ISLE_Reporter`.

2.3.2.2.2.6 Diagnostic Traces

The proxy generates trace records for events that are not related to any particular association and passes them to the interface `ISLE_Trace` provided by the application. It supports the interface `ISLE_TraceControl` to switch tracing on and off. The proxy forwards all requests received via this interface to all associations currently managed.

2.3.2.2.3 Attributes

2.3.2.2.3.1 Protocol ID

Identifies the technology and specific mapping of SLE transfer services to that technology.

2.3.2.2.3.2 Bind Roles Supported

INITIATOR	the proxy is capable to support associations in the role of a bind initiator;
RESPONDER	the proxy is capable to support associations in the role of a bind responder;
ALL	the proxy is capable to support associations in the role of an initiator as well as associations in the role of a responder.

2.3.2.2.4 Behavior and Use

When the method `Configure()` is called on the interface `ISLE_ProxyAdmin`, the proxy checks the information passed and performs all actions required for configuration of the proxy and the communications service. Errors are logged and result in an error code returned to the caller. When the component has been configured successfully, the proxy returns a positive result code, indicating that it is ready for operation. However, it starts processing only when the start method is called on one of the control interfaces `ISLE_Sequential` or `ISLE_Concurrent` (see 2.3.4). This implies that a proxy supporting associations in the responder role starts listening for incoming BIND invocations only after call of the start method.

When the terminate method of the control interface is called, the proxy terminates all threads, if applicable, such that an orderly termination of the application is possible. If any associations are still active when termination is requested, the proxy aborts these associations. A proxy must expect that other proxies using the same communication infrastructure exist on the system and must make sure that their operation is not affected by termination activities.

NOTE – The terminate method is either `TerminateSequential()` of the interface `ISLE_Sequential`, or `TerminateConcurrent()` of the interface `ISLE_Concurrent`, depending on the behavior supported by the proxy.

The proxy provides the method `ShutDown()` to shut it down on its administrative interface `ISLE_ProxyAdmin`. When that method is called it releases all interfaces of other components it still holds, frees all resources, and deletes all internal objects.

2.3.2.2.5 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_ProxyAdmin`	API Proxy	configuration, initialization, dynamic port registration, and shutdown
`ISLE_Concurrent`	Common Control Interfaces	start and termination of operations for concurrent behavior
`ISLE_Sequential`	Common Control Interfaces	start and termination of operations for sequential behavior
`ISLE_AssocFactory`	API Proxy	creation and deletion of associations in the initiator role
`ISLE_TraceControl`	Common Control Interfaces	start and stop of diagnostic traces

2.3.2.2.6 Dependencies

Interface	Defined in Package	Purpose
`ISLE_Reporter`	SLE Application	logging and notification
`ISLE_Trace`	SLE Application	tracing
`ISLE_Locator`	API Service Element	indication of an incoming BIND invocation to the client (not shown in the diagram)
`ISLE_OperationFactory`	SLE Operations	creation of operation objects (not shown in the diagram)
`ISLE_UtilFactory`	SLE Utilities	creation of utility objects (not shown in the diagram)

2.3.2.3 Component Class Association

2.3.2.3.1 General

An object of a class derived from the abstract class Association handles a single data communication association between an SLE service user and an SLE service provider. The class Association defines those aspects of an association, which are independent of the role it plays in the BIND and UNBIND operation. An association is independent of the SLE service type. Service type-specific aspects are handled by the class PDU Translator, to which the association passes all operation invocations and returns for checking and for encoding and decoding.

An association object does not distinguish between the SLE service user role and the SLE service provider role and accepts any PDU that is defined for a given SLE service type. Checking for validity of PDUs for a given role must be performed by the client of the association.

2.3.2.3.2 Responsibilities

2.3.2.3.2.1 Mapping of Port Identifiers

The association maps the logical port identifiers defined by the CCSDS Recommended Standards for SLE transfer services to technology-specific addresses. The mapping is defined in the configuration database.

2.3.2.3.2.2 Processing of SLE Protocol Data Units

Associations accept operation objects holding SLE invocation and return parameters via the interface `ISLE_SrvProxyInitiate`, pass them to the PDU Translator for checking and encoding, and transfer the encoded PDU to the peer proxy. They receive SLE PDUs from the peer proxy, pass them to the PDU Translator for checking and decoding, and forward the resulting operation object to the client via the interface `ISLE_SrvProxyInform`.

2.3.2.3.2.3 Basic SLE Protocol Execution

Association objects implement a basic subset of the state tables defined for SLE services. The state table for associations is specified in section 4.

2.3.2.3.2.4 Authentication

For incoming PDUs, the association determines the required authentication mode defined in the configuration database for the peer application. If authentication is required it uses the interface `ISLE_SecAttributes` provided by the component SLE Utilities to check the credentials transmitted in the PDU. If authentication fails the association ignores the PDU. For outgoing PDUs, the association generates the credentials using the security attributes of the local application in its configuration database.

2.3.2.3.2.5 Monitoring of the State of the Data Communication Connection

The association monitors the state of the data communication connection it uses and informs its client if the connection breaks down. The maximum delay between the failure and the report is specified in the proxy configuration file.

2.3.2.3.2.6 Queuing of Outbound PDUs

The association queues a maximum number of PDUs if these cannot be transmitted immediately. A positive response to a transmission request by the client guarantees that the PDU has been queued for transmission. The maximum size of the queue is defined in the proxy configuration file. If requested by the client (see the interface `ISLE_SrvProxyInitiate`) the association notifies the client when a PDU has actually been transmitted. If the queue of outgoing PDUs is full, the association rejects further transmission requests.

2.3.2.3.2.7 Removal of Transfer Buffer PDUs

On request by the client, the association removes all PDUs of the type TRANSFER-BUFFER for which transmission has not yet started from the queue and releases associated resources. It informs the client whether such PDUs have been discarded.

2.3.2.3.2.8 Limiting Inbound Data Traffic

The association ensures that the number of PDUs received from the network and not yet passed to its client does not exceed a maximum number N1 defined in the proxy configuration file. Of these a maximum number $N2 \leq N1$ are allowed to be PDUs of the type TRANSFER-DATA invocation or TRANSFER-BUFFER invocation. The number N2 is also defined in the proxy configuration file. If either of these limits is exceeded the association does not accept further data from the network making sure that back-pressure is built up.

2.3.2.3.2.9 Logging and Notification

The association and its derived classes generate log records for important events and enter them to the system log using the interface `ISLE_Reporter` provided by the application. For specific events that require immediate attention, the association notifies the application using the method `Notify()` in the interface `ISLE_Reporter`.

2.3.2.3.2.10 Diagnostic Traces

The class Association and its derived classes generate trace records and pass them to the interface `ISLE_Trace` provided by the application. It supports the interface `ISLE_TraceControl` to switch tracing on and off.

2.3.2.3.3 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_SrvProxyInitiate`	API Proxy	passing of SLE PDUs for transfer
`ISLE_TraceControl`	Common Control Interfaces	start and stop of diagnostic traces

2.3.2.3.4 Dependencies

Interface	Defined in Package	Purpose
`ISLE_SrvProxyInform`	API Service Element	passing of SLE PDUs received from the network
`ISLE_Reporter`	SLE Application	logging and notification
`ISLE_Trace`	SLE Application	tracing

2.3.2.4 Internal Class Initiating Association

2.3.2.4.1 General

An initiating association accepts requests to invoke the BIND operation and the UNBIND operation from its local client. If the association receives a BIND or UNBIND invocation PDU from the peer proxy, it aborts the data communication association with the diagnostic ‘protocol error’.

2.3.2.4.2 Responsibilities

2.3.2.4.2.1 Association Establishment

When receiving a BIND invocation from its local client, the initiating association establishes a data communication association with the peer proxy using technology-specific means and transmits the BIND invocation. It completes the association establishment procedure when it receives the BIND return from the peer proxy. If the BIND return PDU contains a positive result, the association is established and the state is set to ‘bound’. If the BIND return PDU carries a negative result, the association is not established and the state is set to ‘unbound’. The association informs its client by forwarding the operation object with the return parameters received from the peer proxy.

The association ensures that the BIND operation is not performed on an established association or during association release and is not re-invoked during association establishment. It also ensures that the BIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

2.3.2.4.2.2 Association Release

When receiving an UNBIND invocation from its local client, the association object forwards the invocation to the peer proxy and initiates termination of the data communication association by technology-specific means. It completes the association release procedure when receiving the UNBIND return, by setting its state to 'unbound' and forwarding the return to its client.

The association ensures that the UNBIND operation is performed only on an established association and is not re-invoked during association release. It also ensures that UNBIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

2.3.2.4.2.3 Access Control

As part of the BIND operation the initiating association locates the peer application in the configuration database of the proxy using the responder identifier in the BIND operation object. When receiving the BIND return it verifies that the responder identifier is the one expected and aborts if that is not the case. This test is performed before authentication, if authentication is required for the peer application.

2.3.2.5 Internal Class Responding Association

2.3.2.5.1 General

A responding association processes BIND invocations received from the peer proxy and responds to UNBIND invocations issued by the peer proxy. It rejects any BIND or UNBIND invocations that might be requested by its local client.

2.3.2.5.2 Responsibilities

2.3.2.5.2.1 Association Establishment

An object of the class Responding Association is created by the API Proxy in order to process a received BIND invocation (see 2.3.2.2.2.4). It performs all initial checks on the BIND invocation defined in this section. It then informs its client via the interface `ISLE_Locator` provided as part of the proxy configuration, passing it a reference to its interface `ISLE_SrvProxyInitiate` and to the operation object holding the BIND invocation parameters.

If the locator interface returns a reference to the interface `ISLE_SrvProxyInform`, the association forwards the BIND invocation via that interface. If the locator returns an error, the association generates a BIND return PDU with a negative result and a diagnostic corresponding to the error. It transmits the PDU to the peer proxy and terminates the data communication association.

The association completes the association establishment procedure when it receives the BIND return from its local client. If the BIND return PDU contains a positive result, the association is established and the state is set to ‘bound’. If the BIND return PDU carries a negative result, the association is not established. In both cases, the association forwards the BIND return to the peer proxy.

If the association receives a BIND invocation from the peer proxy on the data communication association it handles, it aborts the association with the diagnostic ‘protocol error’. The association also ensures that the BIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

2.3.2.5.2.2 Association Release

When receiving an UNBIND invocation from the peer proxy, the association forwards the invocation to its client. It completes the association release procedure when it receives the UNBIND return from its local client.

The association ensures that the UNBIND operation is performed only on an established association and is not re-invoked during association release. It also ensures that UNBIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

2.3.2.5.2.3 Access Control

When receiving a BIND invocation, the association verifies that the initiator is defined as a peer application in the configuration database of the proxy. If that is not the case, it responds with a BIND return containing a negative result and the diagnostic ‘access denied’.

2.3.2.5.2.4 Handling of Service Types and Version Numbers

When receiving a BIND invocation, the association checks that the requested service type is defined in the configuration database of the proxy and that the version number for that type can be supported. If that is not the case, it responds with a BIND return containing a negative result and the appropriate diagnostic.

2.3.2.5.3 Dependencies

Interface	Defined in Package	Purpose
`ISLE_Locator`	API Service Element	indication of an incoming BIND invocation to the client

2.3.2.6 Internal Class PDU Translator

2.3.2.6.1 General

The class PDU Translator translates SLE operation parameters between the syntax used in the API and the syntax required for communication with the peer proxy. The base class handles the SLE operations BIND, UNBIND, and PEER-ABORT, which are identical for all service types. Service-specific operations are handled by the derived classes.

2.3.2.6.2 Responsibilities

2.3.2.6.2.1 Association of Returns with Invocations

The PDU Translator receives operation objects from the association. For invocations of confirmed operations by the local client, it stores a reference to the operation object until the return from the peer proxy arrives or the association is aborted. When receiving a PDU that contains a return, the PDU Translator locates the operation object holding the invocation by means of the invocation identifier. The PDU Translator verifies that the invocation and return are of the same operation type. If it cannot locate the invocation, it informs the association, which aborts with the diagnostic 'unsolicited invocation identifier'.

NOTE – The processing of invocations of confirmed operations received from the peer proxy is described in 2.3.2.6.2.3.

It is noted that the confirmed operations BIND and UNBIND do not carry an invocation identifier. Because only a single return can be outstanding for these operations at any time, association of the return with the invocation is possible without the invocation identifier.

2.3.2.6.2.2 Encoding of PDUs

A PDU Translator extracts the invocation or return parameters from the operation objects passed by the association, encodes them as required by the technology mapping, builds the protocol data unit, and passes it back to the association for transmission.

2.3.2.6.2.3 Decoding of PDUs

The PDU Translator decodes PDUs received from the peer proxy and extracts the operation parameters. For invocations, the PDU Translator creates an operation object using the interface `ISLE_OperationFactory`, stores the invocation parameters to this object, and passes it to the association for further processing. For returns, it stores the return parameters to the operation object, which holds the associated invocation.

2.3.2.7 Internal Class <SRV> PDU Translator

A class <SRV> PDU Translator exists for every SLE service type supported by the proxy. For operation objects and PDUs passed to the PDU Translator, the class checks whether these are defined for the service type. If the operation is defined for the service type, the class performs encoding and decoding for these operations as described in 2.3.2.6.

2.3.2.8 Proxy Configuration Database

2.3.2.8.1 General

Operation of the API Proxy in a specific deployment environment is controlled by parameters in a configuration database. The structure of this database is implementation specific. It could consist of one or more files or could be implemented using directory systems or some management database. The configuration file passed to the proxy as part of the configuration can contain the complete database or only a reference that enables the proxy to access the database.

Also, the content of the database is largely implementation specific. Information, which must be part of the configuration database, is presented in figure 2-5. The objects shown in the figure are not complete. Information objects not shown in the figure are represented by the attribute 'Configuration Parameters' of the class Proxy Config Database. A complete list of required objects may be found in section 3.

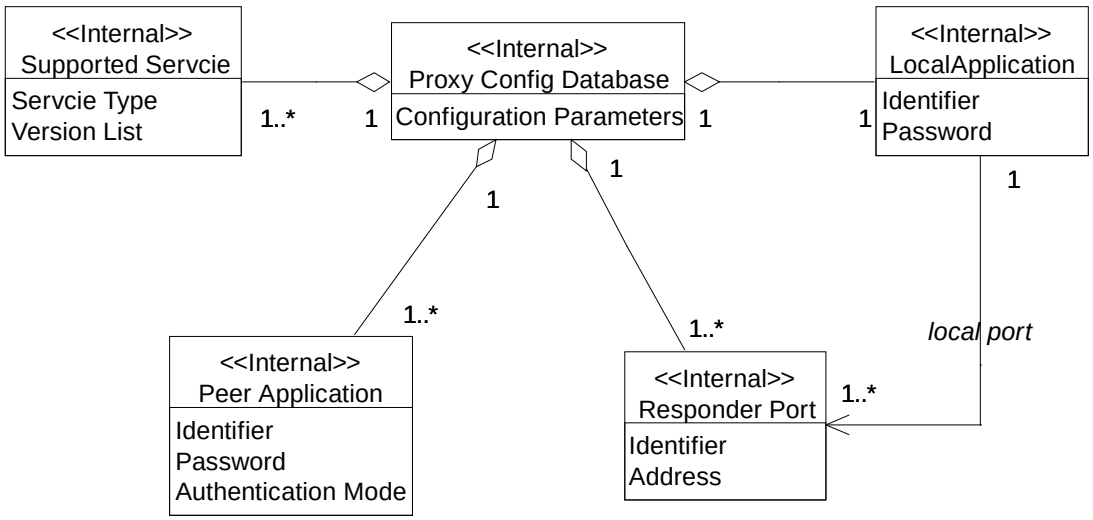

Figure 2-5: Configuration Database of the Proxy

2.3.2.8.2 Local Application

The database contains information concerning the local SLE application. The information includes the identifier (user name) and a password for authentication. For proxies supporting associations in the responder role, the database also holds information related to local ports, on which the proxy shall accept incoming BIND invocations. The number of ports that can be supported is implementation defined.

2.3.2.8.3 Peer Application

The database contains a list of peer applications, which are allowed to access the system. The list includes service users that may access a service provider and service providers, which the local application may use. For every peer application, the database contains the identifier, the required authentication mode (no authentication, authentication for BIND only, authentication of all PDUs) and a password for authentication.

2.3.2.8.4 Port

For all responder ports (local and remote) the database contains a logical port identifier and the technology-specific address information associated with that name.

2.3.2.8.5 Supported Services

The database contains a list of the service types that are supported by all API components in an installation, and for each type, the list of version numbers that are supported by all API components.

2.3.2.8.6 Interfaces Defined by the Package

Name	Description
`ISLE_ProxyAdmin`	The interface is provided to configure and initialize the proxy component passing it the pointers to interfaces of other components it needs operationally. In addition, the interface comprises the methods for port registration and de-registration and for shutting down of the proxy.
`ISLE_AssocFactory`	The interface supports creation of initiating association objects for a specified SLE transfer service type. A pointer to the client interface must be passed to the creation function. The interface also provides a method to request the proxy to destroy an association object that is no longer needed.
`ISLE_SrvProxyInitiate`	The interface provides methods to pass SLE operation invocations and returns for transmission. In addition, it supports the features to request reporting of actual transfer of a PDU, and to discard PDUs of the type TRANSFER-BUFFER. The interface is identical for all association roles (initiator and provider) and all SLE service types.

2.3.3 PACKAGE API SERVICE ELEMENT

2.3.3.1 Overview

The API Service Element implements functionality related to SLE transfer service provisioning, which can be clearly separated from service production. It provides support for SLE service provider applications and for SLE service user applications. The structure of the API Service Element is shown in figure 2-6.

The component class API Service Element is responsible for configuration, initialization, and management of the component. It provides an interface to the application to create and delete service instances (`ISLE_SIFactory`) and to the proxy to locate service instances when receiving a BIND invocation (`ISLE_Locator`).

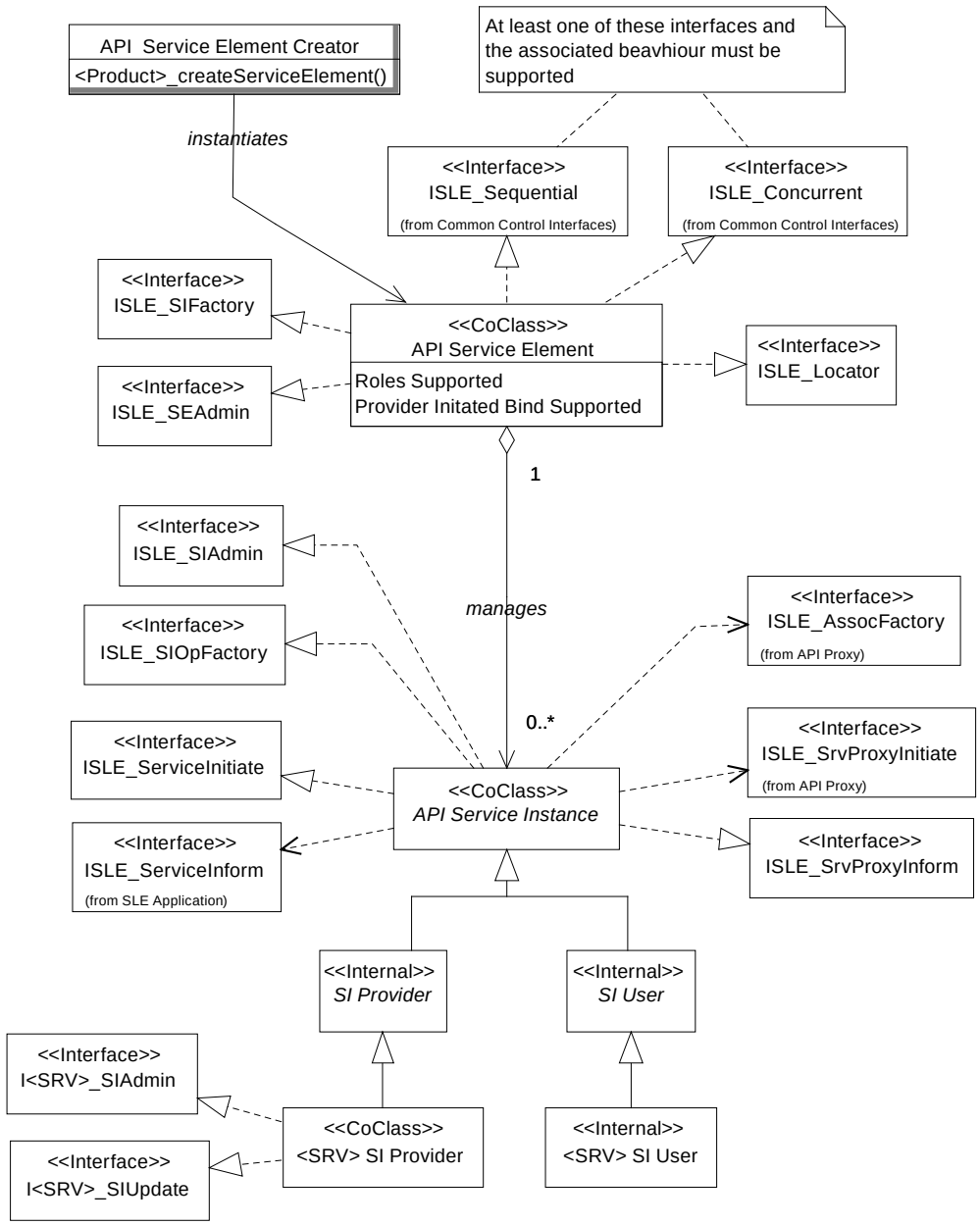

Figure 2-6: Structure of the Package API Service Element

Individual service instances are handled by the class API Service Instance. During periods in which an SLE service user and an SLE service provider communicate, the service instance is linked with an association object in the component API Proxy. It communicates with the association via the interface `ISLE_SrvProxyInitiate` and the complementary interface `ISLE_SrvProxyInform`. With the application it communicates via the interface `ISLE_ServiceInitiate` and the complementary interface `ISLE_ServiceInform`.

Service instances are distinguished according to the application role they support. The class SI Provider supports SLE service provider applications and the class SI User supports SLE

service user applications. While the behavior of these classes and the operations they support differ, the externally visible interfaces are identical.

The classes API Service Instance, SI Provider, and SI User are abstract. Service instance objects support a specific SLE service type represented by the classes <SRV> SI Provider and <SRV> SI User. The class <SRV> SI User does not export any new interfaces, while service instances in the provider role support additional service type-specific interfaces for configuration (`I<SRV>_SIAdmin`) and for update of service parameters (`I<SRV>_SIUpdate`).

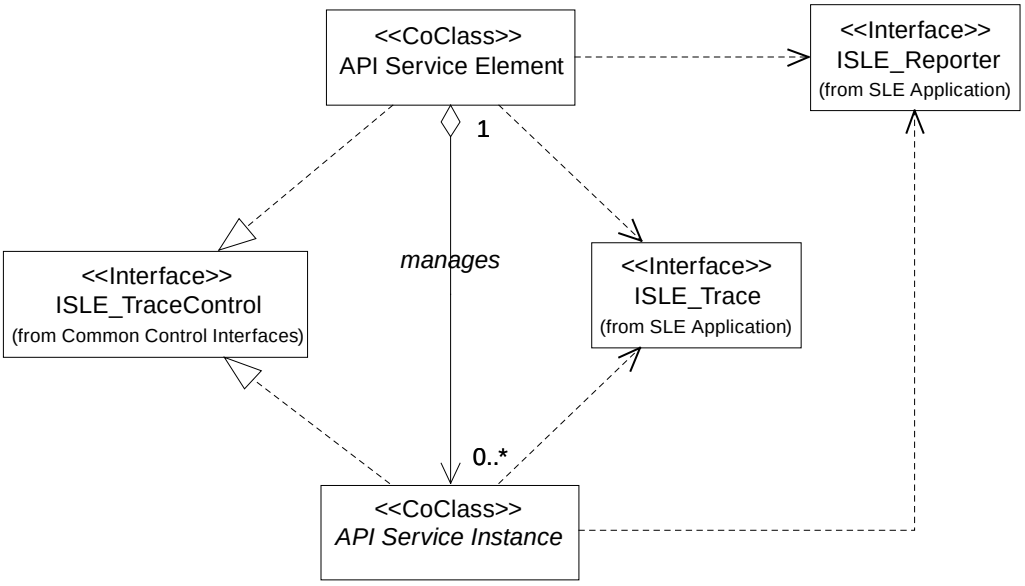

Figure 2-7: Reporting and Tracing by the Service Element

The service element and service instances support logging and diagnostic traces using the interfaces `ISLE_Reporter` and `ISLE_Trace` provided by the application. Diagnostic Traces can be switched on and off via the interface `ISLE_TraceControl` exported by the API Service Element and by the API Service Instance.

NOTE – All classes in the package API Service Element use the interfaces of operation objects and of utility objects. This fact is not specifically mentioned in the following description.

2.3.3.2 Component Class API Service Element

2.3.3.2.1 General

The component class API Service Element provides management of service instances. An implementation of the component can support service instances for the SLE service provider role and for the SLE service user role or only for one of the roles as indicated by the attribute 'Roles Supported'. Support for user-initiated binding of service instances is mandatory, while support for provider-initiated binding is an option. Its support is indicated by the attribute 'Provider Initiated Bind Supported'.

NOTE – This version of the Recommended Practice does not support provider-initiated binding, see 1.2.2 item b). The option and the attribute 'Provider Initiated Bind Supported' are foreseen to allow later extension.

A single instance of this class exists within an instance of the API Service Element component.

2.3.3.2.2 Responsibilities

2.3.3.2.2.1 Configuration and Initialization of the API Service Element Component

After creation, the API Service Element must be configured and initialized using the interface `ISLE_SEAdmin`. All static configuration parameters needed for this purpose are specified in the configuration database defined in 2.3.3.8.

2.3.3.2.2.2 Control of Proxies

The service element can use several proxies distinguished by the 'Protocol ID' of the proxy. The interface `ISLE_SEAdmin` provides a method to link proxies to the service element after configuration. The service element starts and terminates operation of all linked proxies when its own operation is started or terminated.

For service instances that initiate the BIND operation, the service element selects the proxy to use by a table in its configuration database, which associates the peer port identifier with the Protocol ID supported by the proxy.

2.3.3.2.2.3 Management of Service Instances

The service element creates and initializes service instances for a specified service type and a specified role on request of the application via the interface `ISLE_SIFactory`. If the service element does not support the requested service type or role, it rejects the request. If the service instance shall initiate binding, the application must additionally specify the version number of the service type.

The service element keeps a reference to the service instances created until the application requests it to destroy the service instance. If the service instance is still bound at that time, the service element rejects the request. Otherwise, it releases all resources that are allocated to the service instance and performs all actions required to delete the service instance.

2.3.3.2.2.4 Location of Service Instances

The service element provides the interface `ISLE_Locator` to the proxy to locate requested service instances when the proxy receives a BIND invocation from the peer system. It uses the service instance identifier in the BIND invocation passed by the proxy to find the service instance. If the service instance has been created by the application, the service element verifies that this service instance is not already bound and that the BIND invocation parameters are consistent with the configuration of the service instance. If all checks are passed, it links the service instance with the association and returns a reference to the interface `ISLE_SrvProxyInform` of the service instance to the proxy. Otherwise, it returns an error, instructing the proxy to reject the BIND invocation.

2.3.3.2.2.5 Access Control

For incoming BIND invocations the service element verifies that the initiator identifier in the BIND invocation matches the peer identifier defined for this service instance. If that is not the case, it rejects the request and generates an access violation alarm.

2.3.3.2.2.6 Logging and Notification

The service element generates log records for important events and enters them to the system log using the interface `ISLE_Reporter` provided by the application. For events that require immediate attention, the service element notifies the application using the method `Notify()` in the interface `ISLE_Reporter`.

2.3.3.2.2.7 Diagnostic Traces

The service element generates trace records for events that are not related to any particular service instance and passes them to the interface `ISLE_Trace` provided by the application. It supports the interface `ISLE_TraceControl` to switch tracing on and off. The service element forwards all requests received via this interface to all service instances currently managed, and, if requested by the caller, to all proxies that it controls.

NOTE – The interface `ISLE_TraceControl` of the service element allows setting of the trace level on a global scope. Individual setting of the trace level of each service instance is possible using the interface `ISLE_TraceControl` of the service instance.

2.3.3.2.3 Attributes

2.3.3.2.3.1 Roles Supported

USER	the service element supports service instances in the role of an SLE service user;
PROVIDER	the service element supports service instances in the role of an SLE service provider;
ALL	the service element supports service instances in the role of a user as well as service instances in the role of a provider.

2.3.3.2.3.2 Provider Initiated Bind Supported

Indicates whether the service element supports provider-initiated binding of service instances.

NOTE – This version of the Recommended Practice does not support provider-initiated binding; see 1.2.2 item b). The option and the attribute 'Provider Initiated Bind Supported' are foreseen to allow later extension.

2.3.3.2.4 Behavior and Use

When the method `Configure()` is called on the interface `ISLE_SEAdmin()`, the service element checks the information passed, and performs all actions required for configuration of the component. Errors are logged and result in an error code returned to the caller. When the component has been configured successfully, the service element returns a positive result code. Following configuration of the component, the service element must be linked with the proxies it will use. For this purpose, the service element provides the method `AddProxy()` in its administrative interface.

The service element starts processing when the start method is called on one of the control interfaces `ISLE_Sequential` or `ISLE_Concurrent` (see 2.3.4). It then also starts processing of all linked proxies using the interface selected for control of the proxy.

When the terminate method is called via the control interface, the service element terminates all threads, if applicable, such that an orderly termination of the application is possible. If any service instances are still active when termination is requested, the service element instructs them to abort the association. Finally the service element terminates processing of all linked proxies.

NOTE – The terminate method is either `TerminateSequential()` of the interface `ISLE_Sequential`, or `TerminateConcurrent()` of the interface `ISLE_Concurrent`, depending on the behavior supported by the service element.

The service element provides the method `ShutDown()` to shut it down on its administrative interface `ISLE_SEAdmin`. When that method is called it releases all interfaces of other components it still holds, frees all resources, and deletes all internal objects.

2.3.3.2.5 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_SEAdmin`	API Service Element	configuration, initialization, and shutdown
`ISLE_Concurrent`	Common Control Interfaces	start and termination of operations for concurrent behavior
`ISLE_Sequential`	Common Control Interfaces	start and termination of operations for sequential behavior
`ISLE_SIFactory`	API Service Element	creation and deletion of service instances
`ISLE_Locator`	API Service Element	location of service instances requested by incoming BIND invocations
`ISLE_TraceControl`	Common Control Interfaces	start and stop of diagnostic traces

2.3.3.2.6 Dependencies

Interface	Defined in Package	Purpose
`ISLE_Reporter`	SLE Application	logging and notification
`ISLE_Trace`	SLE Application	tracing
`ISLE_OperationFactory`	SLE Operations	creation of operation objects (not shown in the diagram)
`ISLE_UtilFactory`	SLE Utilities	creation of utility objects (not shown in the diagram)
`ISLE_Concurrent`	Common Control Interfaces	start and termination of proxies for concurrent behavior (not shown in the diagram)
`ISLE_Sequential`	Common Control Interfaces	start and termination of proxies for sequential behavior (not shown in the diagram)
`ISLE_TraceControl`	Common Control Interfaces	start and stop of diagnostic traces of proxies (not shown in the diagram)

2.3.3.3 Component Class API Service Instance

2.3.3.3.1 General

An object of a class derived from the abstract class API Service Instance handles a single SLE transfer service instance. The class API Service Instance defines those aspects of a service instance, which are independent of the user and provider role and independent of a specific service type.

2.3.3.3.2 Responsibilities

2.3.3.3.2.1 Configuration of the Service Instance

The service instance exports the interface `ISLE_SIAdmin` to set common configuration parameters after creation. When all parameters have been set, the method `ConfigCompleted()` must called. The service instance then checks its configuration for completeness and consistency.

2.3.3.3.2.2 Control of Initiating Associations

If the service instance initiates binding, it creates the initiating association via the interface `ISLE_AssocFactory` exported by the component API Proxy. It selects the proxy instance from the mapping table in the configuration database of the service element, using the responder port identifier as a key. The service instance requests the proxy to destroy the association when it is no longer needed.

Implementations might create the association after configuration and keep it for the complete lifetime of the service instance or create a new association for every BIND invocation.

2.3.3.3.2.3 Processing of SLE Protocol Data Units

The service instance receives operation objects holding SLE PDUs from the application via the interface `ISLE_ServiceInitiate`. It verifies that the PDUs are valid in the current state and checks the parameters for completeness, consistency, and range. If all checks are passed, the service instance passes the operation objects to the association for transfer via the interface `ISLE_SrvProxyInitiate`. With a positive result code returned to the application, the service instance guarantees that the PDU has been accepted by the proxy.

The service instance receives operation objects holding SLE PDUs from the association via the interface `ISLE_SrvProxyInform`. It verifies that the PDUs are valid in the current state and checks the parameters for completeness, consistency, and range. If all checks are passed and the operation is not handled by the service instance itself, it passes the operation objects to the application via the interface `ISLE_ServiceInform`.

2.3.3.3.2.4 SLE Protocol Execution

The service instance enforces conformance to the state tables defined for SLE services to the extent that these are independent of service production. The state tables processed by service instances are specified in section 4.

2.3.3.3.2.5 Management of Invocation Identifiers

The service instance assigns unique invocation identifiers to operation objects for confirmed SLE operations. For invocations of confirmed operations received from the proxy, the service instance verifies that the invocation identifier is unique for all operations to which the application has not yet responded. If the service instance detects a duplicate invocation identifier, it responds with a return containing a negative response and the appropriate diagnostic.

It is noted that the confirmed operations BIND and UNBIND do not carry an invocation identifier and must be excluded from these checks.

2.3.3.3.2.6 Timeout Monitoring for Operation Returns

For confirmed operations invoked by the local application or by the service instance itself, the service instance ensures that a return is received within a timeout defined as a configuration parameter. If no return arrives within the specified timeout, it aborts the association.

2.3.3.3.2.7 Pre-setting of Operation Object Parameters

The service instance provides an interface for creation of operation objects for the service type supported. It uses the interface `ISLE_OperationFactory` exported by the component SLE Operations to create these objects and initializes the parameters of the operation objects according to its own configuration.

2.3.3.3.2.8 Logging and Notification

The class API Service Instance and its derived classes generate log records for important events and enter them to the system log using the interface `ISLE_Reporter` provided by the application. For events that require immediate attention, the service instance notifies the application using the method `Notify()` in the interface `ISLE_Reporter`.

2.3.3.3.2.9 Diagnostic Traces

The class API Service Instance and its derived classes generate trace records and pass them to the interface `ISLE_Trace` provided by the application. It supports the interface `ISLE_TraceControl` to switch tracing on and off. If requested by the caller, the service instance forwards the request to the associations, which it is using.

2.3.3.3.3 Behavior

For handling of errors, the service instance applies the following rules:

a) If SLE PDUs received from the application are not valid in the current state or fail to pass any of the other checks the service instance applies, it returns an error code to the method that passed the PDU.

b) If a PDU received from the application is rejected by the association, the service instance returns the result code received from the association to the application. If the result code indicates that the queuing capacity of the association is exceeded, the service instance aborts the association. Because of the flow control mechanisms built into the API, queue overflow cannot be caused by transfer of space link data units. It can only happen because of excessive generation of other events related to the production process or excessively high status reporting frequencies. In these cases the application would have no other option to handle the problem.

c) If SLE PDUs received from the association are not valid in the current state or fail to pass any of the other checks the service instance applies, the service instance proceeds as follows:

   1) if the problem is due to a misbehavior of the association, it returns an error code to the method that passed the PDU;

   2) if the PDU is an invocation of a confirmed operation, the service instance sets the result of the operation object to ‘negative’, inserts the appropriate diagnostic, and forwards it to the association for transfer;

   3) otherwise, the service instance aborts the association with the appropriate diagnostic.

2.3.3.3.4 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_SIAdmin`	API Service Element	configuration of the service instance
`ISLE_SIOpFactory`	API Service Element	creation and initialization of operation objects
`ISLE_ServiceInitiate`	API Service Element	passing of SLE PDUs from the application to the service instance
`ISLE_SrvProxyInform`	API Service Element	passing of SLE PDUs received from the association
`ISLE_TraceControl`	Common Control Interfaces	start and stop of diagnostic traces

2.3.3.3.5 Dependencies

Interface	Defined in Package	Purpose
`ISLE_AssocFactory`	API Proxy	creation and deletion of associations in the initiator role
`ISLE_ServiceInform`	SLE Application	passing of SLE PDUs to the application
`ISLE_SrvProxyInitiate`	API Proxy	passing of SLE PDUs to the association for transfer
`ISLE_OperationFactory`	SLE Operations	creation of operation objects (not shown in the diagram)
`ISLE_TraceControl`	Common Control Interfaces	start and stop of diagnostic traces of the association (not shown in the diagram)

2.3.3.4 Internal Class SI User

2.3.3.4.1 General

The class SI User defines those aspects of a service instance in the user role, which are independent of a specific SLE service type. This class does not export any additional interfaces.

2.3.3.4.2 Responsibilities

2.3.3.4.2.1 Processing of SLE Protocol Data Units

The service instance verifies that PDUs received from the association or from the application are compatible with the user role, the service type supported, and with the version number of the service.

2.3.3.4.2.2 Buffering for Return Services

For return services, the service instance accepts the TRANSFER-BUFFER operation object from the association, extracts the TRANSFER-DATA and SYNC-NOTIFY operation objects, and passes them to the application in the sequence they have been stored in the TRANSFER-BUFFER operation object. The service instance verifies that the buffer received only contains PDUs for which buffering shall be applied.

2.3.3.4.2.3 Flow Control for Forward Services

For forward services, the service instance provides flow control for TRANSFER-DATA invocations. When a maximum number of TRANSFER-DATA invocations have been queued by the association and not yet transmitted, the service instance returns a code to the application requesting it to suspend data transfer. It informs the application when data transmission can be resumed via the method `ResumeDataTransfer()` of the interface `ISLE_ServiceInform`. The number of invocations that can be queued is defined by the implementation or can be set in the configuration database of the service element.

2.3.3.5 Internal Class SI Provider

2.3.3.5.1 General

The class SI Provider defines those aspects of a service instance in the provider role, which are independent of a specific SLE service type. This class does not export any additional interfaces.

2.3.3.5.2 Responsibilities

2.3.3.5.2.1 Processing of SLE Protocol Data Units

The service instance verifies that PDUs received from the association or from the application are compatible with the provider role, the service type supported, and with the version number of the service. It extracts the applicable version number from the BIND invocation received from the proxy.

2.3.3.5.2.2 Buffering for Return Services

For return services the service instance handles the transfer buffer defined by CCSDS Recommended Standards for return link services. For this purpose, it uses the operation object for the TRANSFER-BUFFER operation. The service instance adds invocations of the operations TRANSFER-DATA and SYNC-NOTIFY received from the application to the TRANSFER-BUFFER operation object, and forwards it to the association when the buffer is full. The size of the buffer is a parameter passed to the service instance as part of its configuration.

2.3.3.5.2.3 Buffering in the Delivery Modes Timely Online and Complete Online

For the delivery modes ‘timely online’ and ‘complete online’, the service instance handles the release timer as defined by the CCSDS Recommended Standards for return link services. The service instance starts the release timer when inserting the first PDU into the transfer buffer. When the buffer is full, when the release timer expires, or when the last PDU appended to the buffer is an ‘end of data’ SYNC-NOTIFY operation, the service instance forwards the transfer buffer content to the association for transfer in the form of a TRANSFER-BUFFER invocation.

NOTE – The term ‘TRANSFER-BUFFER invocation’ corresponds to a transmission request for the transfer buffer, not an SLE operation.

2.3.3.5.2.4 Buffering in the Delivery Mode Timely Online

For the delivery mode timely online the service instance additionally handles discarding of buffers as defined by the CCSDS Recommended Standards for SLE return link services. When a transfer buffer is due for transmission, it performs the following steps:

a) If the association did not yet notify transmission of the previous TRANSFER-BUFFER invocation, the service instance requests the association to discard the queued (previous) TRANSFER-BUFFER invocation.

b) If the result returned by the association confirms that the association has actually discarded a TRANSFER-BUFFER invocation, the service instance inserts a notification 'data discarded due to excessive backlog' at the beginning of the transfer buffer before forwarding it to the association.

2.3.3.5.2.5 Flow Control for Complete Online and Offline Delivery Modes

In the delivery modes complete online and offline, the service instance provides flow control for TRANSFER-DATA invocations. If the transfer buffer fills up and a previously sent TRANSFER-BUFFER invocation has not yet been transmitted by the association, the service instance returns a code to the application requesting it to suspend data transfer. It informs the application when data transmission can be resumed via the method `ResumeDataTransfer()` of the interface `ISLE_ServiceInform`.

2.3.3.5.2.6 GET-PARAMETER Operation

The service instance performs the operation GET-PARAMETER without involving the application. It stores the current value of the requested parameter into the operation object and forwards it to the association for transfer. It is noted that the GET-PARAMETER operation is service specific and the derived service-specific class must be involved.

2.3.3.5.2.7 Status Reporting

The service instance performs the operation SCHEDULE-STATUS-REPORT without involving the application. It handles the report timer, generates status reports when needed, and forwards them to the association for transfer. The status reports contain the current values of the service parameters. It is noted that the STATUS-REPORT operation is service specific and the derived service-specific class must be involved.

2.3.3.5.2.8 Service Provisioning Period

The service instance accepts a BIND invocation only within the scheduled provision period. If the state of the service instance is not 'unbound' at the end of the provision period it aborts the association. It informs the application of the end of the provisioning period via the interface `ISLE_ServiceInform`.

For special purposes, the service provision period can be declared as infinite by setting the start and end times to NULL. If that is done, the service instance assumes that the provision period starts as soon as configuration is completed and never terminates.

2.3.3.6 Internal Class <SRV> SI User

A class <SRV> SI User exists for every service type supported by the service element. It ensures that the SLE PDUs passed by the application and by the association are supported by the service type and handles the service-specific operation objects.

2.3.3.7 Component Class <SRV> SI Provider

2.3.3.7.1 General

A class <SRV> SI Provider exists for every service type supported by the service element.

2.3.3.7.2 Responsibilities

2.3.3.7.2.1 Processing of SLE Protocol Data Units

The service instance ensures that the SLE PDUs passed by the application and by the association are supported by the service type and handles the service-specific operation objects.

2.3.3.7.2.2 Service Specific Configuration

The service instance provides an interface to define the service-specific configuration parameters. This interface is defined by the relevant supplemental Recommended Practice for the service-specific API.

2.3.3.7.2.3 Update of Service Parameters

If defined by the relevant supplemental Recommended Practice for the service-specific API, the service instance provides an interface to update the values of service parameters used for the GET-PARAMETER return and for status reports.

2.3.3.7.2.4 GET-PARAMETER and Status Reporting

The service instance generates the service-specific GET-PARAMETER returns and status reports.

2.3.3.7.2.5 Handling of Service Parameters after UNBIND or Abort

Following completion of the UNBIND operation with the unbind-reason ‘suspend’ or after an abort, the service instance sets the configuration parameters as defined for the specific service type.

2.3.3.7.3 Exported Interfaces

Interface	Defined in Package	Purpose
`I<SRV>_SIAdmin`	Service Supplement	Configuration of the service instance
`I<SRV>_SIUpdate`	Service Supplement	Update of service parameters

2.3.3.8 Service Element Configuration Database

The execution of the API Service Element is controlled by parameters in a configuration database. The structure of this database is implementation specific. It could consist of one or more files or could be implemented using directory systems or some management database. The configuration file passed to the service element as part of the configuration can contain the complete database or only a reference that enables the service element to access the database.

Also the content of the database is largely implementation specific. Elements required by this Recommended Practice include

a) a table associating port identifiers with protocol identifiers to select the proxy for outgoing BIND invocations;

b) for a service element supporting the provider role, the minimum and maximum reporting cycle supported.

2.3.3.9 Interfaces Defined by the Package

Name	Description
`ISLE_SEAdmin`	The interface is provided to configure and initialize the service element component passing it the pointers to interfaces of other components it needs. In addition, the interface comprises the methods for linking proxy components and for shutting down the service element.
`ISLE_SIFactory`	The interface allows creation of service instances for a specified service type and with a specified role (service user or service provider). It also provides a method to request deletion of a service instance object that is no longer needed.
`ISLE_Locator`	The locator interface is used to locate a service instance, using the parameters of a BIND invocation, and to link it with an association object.
`ISLE_SIAdmin`	The interface provides the methods needed to set common configuration parameters for a service instance and to complete configuration. Service type-specific configuration parameters must be set by the interface specified for that type.
`ISLE_SIOpFactory`	The interface allows creation of operation objects for the service type supported by the service instance, and initialization of invocation parameters according to the configuration of the service instance.
`ISLE_SrvProxyInform`	The interface provides the methods to pass SLE operation invocations and returns received from the peer proxy. In addition, it supports reporting of actual transfer of a PDU.
`ISLE_ServiceInitiate`	The interface provides the methods to pass SLE operation invocations and returns from the application to a service instance and to read the state of the service instance.

2.3.4 PACKAGE COMMON CONTROL INTERFACES

2.3.4.1 Overview

In order to ensure substitutability, handling of multiple flows of control must be well defined at interfaces between components. This specification defines two behaviors:

a) sequential behavior, in which a single flow of control at a time may pass an interface;

b) concurrent behavior, in which multiple flows of control can pass an interface concurrently.

NOTES

1 Multiple flows of control are frequently implemented by in-process threads but can also be provided by interrupt handlers or other operating system features. In this Recommended Practice, the term ‘thread’ is used in a broader sense referring to any kind of flow of control.

2 The terms ‘sequential’ and ‘concurrent’ have been adopted from the characteristics defined in UML for operations. However, the meaning of ‘sequential’ is slightly more restrictive and the term ‘concurrent’ as used in this Recommended Practice maps to ‘concurrent or guarded’ in UML.

These behaviors are defined to more detail in 2.3.4.2 and 2.3.4.3. The behavior must be respected by the supplier of an interface and by the client of an interface. The same behavior is assumed for complementary interfaces. Components are required to support at least one of these behaviors but can support both.

A component providing a specific behavior for its interfaces exports an associated control interface to start and terminate processing of the component. These control interfaces are defined by the package Common Control Interfaces. The interface `ISLE_Sequential` is supported by components providing sequential behavior and the interface `ISLE_Concurrent` is supported by components providing concurrent behavior.

For the sequential interface behavior, this Recommended Practice also defines interfaces by which the client offers means for components to wait for external events and to handle timers. Components providing concurrent behavior are expected to handle external events and timers internally.

In addition, this package defines an interface to start and stop diagnostic traces, which is implemented by all components providing that option.

2.3.4.2 Sequential Behavior

2.3.4.2.1 Definitions

A component providing sequential behavior on an interface provided to the application or to a higher layer API component ensures that methods of the complementary client interfaces are only called in a thread that originates from a client call. Use of multiple threads by the component is not excluded, but the component must guarantee that no thread started by the component itself or by any lower layer component enters client code.

Because of these restrictions, components providing sequential interface behavior cannot wait for external events or timers without blocking the client. Therefore, the client provides specific interfaces for monitoring of events and handling of timers on behalf of the component.

The application or API components using interfaces of lower layer API components with sequential behavior ensure that methods of these interfaces are invoked sequentially. Use of multiple threads by clients is not excluded, but access to the interface must be strictly serialized.

2.3.4.2.2 Sequential Control Interface

2.3.4.2.2.1 General

The elements of the sequential control interface are shown in figure 2-8. The interface `ISLE_Sequential` must be implemented by the controlled component. The client of the interface, the 'controller', provides services to the controlled component to listen for external events and to handle timers. In the model, these services are described by the component classes Event Monitor and Timer Handler. The controlled component implements the interfaces that shall be called when an external event is detected (`ISLE_EventProcessor`) or a timer expires (`ISLE_TimeoutProcessor`). In the model, these interfaces are provided by the component classes Event Processor and Timeout Processor. The controlled component can use one or more instances of these classes and of the associated interfaces.

The component class 'Controlled Component' is actually a placeholder for a component class that provides the interface `ISLE_Sequential`. This can be the component class API Proxy or the component class API Service Element. Processing of the methods to start and terminate operation is described in the packages API Proxy and API Service Element.

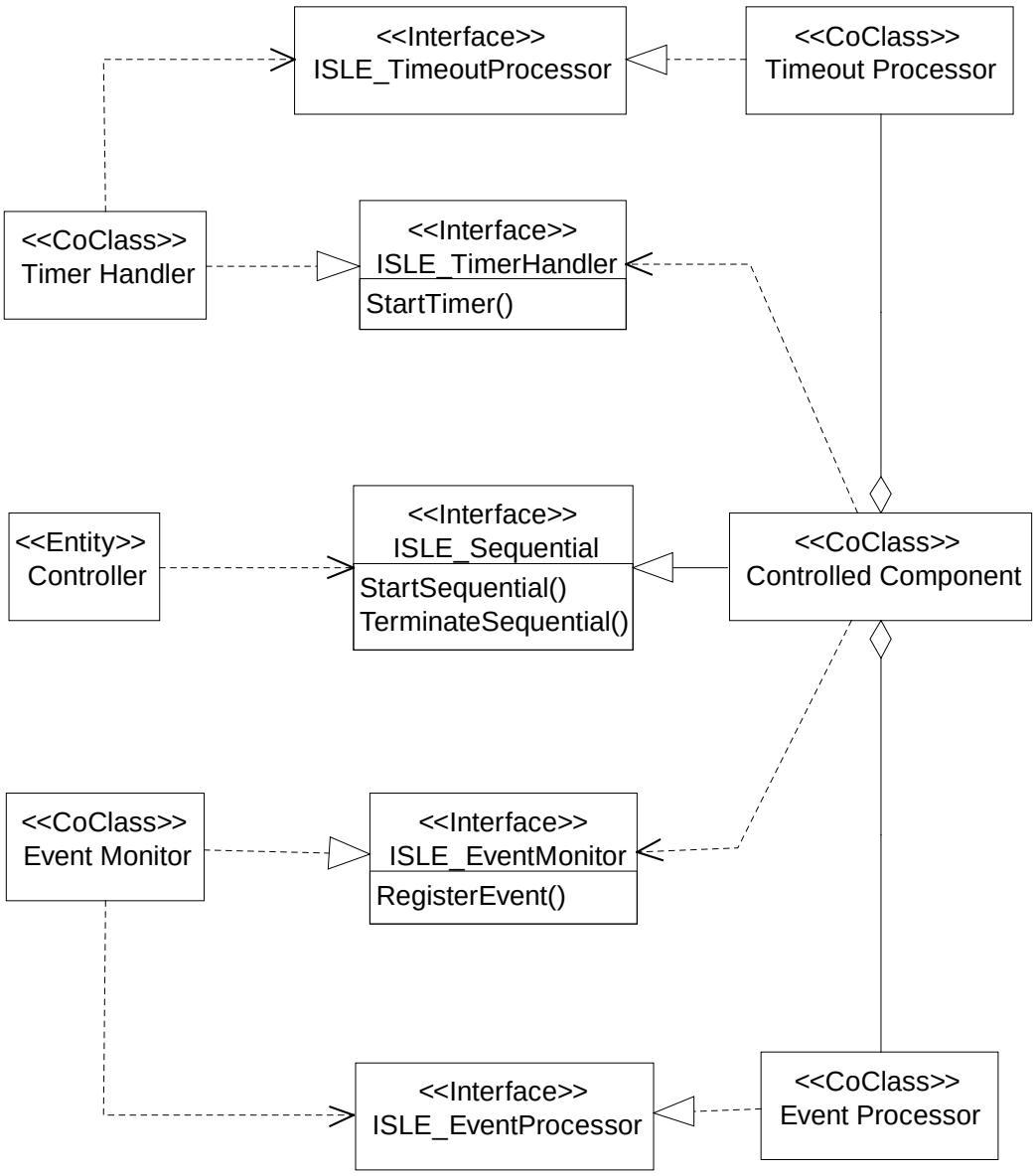

Figure 2-8: Sequential Control Interface Component Class Controlled Component

2.3.4.2.2.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_Sequential`	Common Control Interfaces	start and termination of processing and supply of interfaces for monitoring of external events and timer handling

2.3.4.2.2.3 Dependencies

Interface	Defined in Package	Purpose
`ISLE_EventMonitor`	Common Control Interfaces	monitoring of external events
`ISLE_TimerHandler`	Common Control Interfaces	timer Handling

2.3.4.2.3 Component Class Event Monitor

2.3.4.2.3.1 General

The event monitor supports registration of external events it shall monitor, together with a reference to the interface `ISLE_EventProcessor`. When a registered event occurs, the event monitor calls the method `ProcessEvent()` of the interface `ISLE_EventProcessor`. Events can also be removed from the event monitor. If the event monitor is no longer able to handle an event, it informs the event processor, using the method `MonitorAbort()`.

2.3.4.2.3.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_EventMonitor`	Common Control Interfaces	monitoring of external events

2.3.4.2.3.3 Dependencies

Interface	Defined in Package	Purpose
`ISLE_EventProcessor`	Common Control Interfaces	processing of external events

2.3.4.2.4 Component Class Event Processor

2.3.4.2.4.1 General

The event processor processes an event detected by the event monitor as required for the component.

2.3.4.2.4.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_EventProcessor`	Common Control Interfaces	processing of external events

2.3.4.2.5 Component Class Timer Handler

2.3.4.2.5.1 General

The timer handler supports starting of timers, together with a reference to the interface `ISLE_TimeoutProcessor`. When the timer expires, the timer handler calls the method `ProcessTimeout()` of the interface `ISLE_TimeoutProcessor`. Running timers can be cancelled. If the timer handler is no longer able to support a running timer, it informs the timeout processor, using the method `HandlerAbort()`.

2.3.4.2.5.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_TimerHandler`	Common Control Interfaces	timer handling

2.3.4.2.5.3 Dependencies

Interface	Defined in Package	Purpose
`ISLE_TimeoutProcessor`	Common Control Interfaces	processing of a timeout

2.3.4.2.6 Component Class Timeout Processor

2.3.4.2.6.1 General

The timeout processor processes a timeout detected by the timer handler as required for the component.

2.3.4.2.6.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_TimeoutProcessor`	Common Control Interfaces	processing of a timeout

2.3.4.3 Concurrent Behavior

2.3.4.3.1 Definitions

Components providing concurrent interface behavior are able to handle concurrent calls to the methods of the interface by several flows of control. Components can guard methods of the interface to achieve sequential semantics, but this fact is not visible to clients.

NOTE – When multiple threads access object data or global data within the component, at least access to these data must be serialized using some kind of guard.

It is expected that components providing concurrent interface behavior use multiple threads of control internally. Therefore, they are able to wait for external events and timers without affecting their clients.

Clients of an interface with concurrent behavior must expect that the methods of the complementary interface are called by concurrent flows of control.

When SLE protocol data units are passed across an interface with concurrent characteristics, sequence preservation is not guaranteed when PDUs are passed in one direction by more than one thread. Therefore, this Recommended Practice foresees sequence counts that allow the receiver to re-sequence PDUs. Although the actual need for sequence counting depends on the implementation of a multi-threaded component, this Recommended Practice requires that sequence counts be always used on an interface with concurrent behavior.

2.3.4.3.2 Concurrent Control Interface

The concurrent control interface is shown in figure 2-9.

The component class 'Controlled Component' in figure 2-9 is actually a placeholder for a component class that provides the interface `ISLE_Concurrent`. This can be the component class API Proxy or the component class API Service Element. Processing of the methods to start and terminate operation is described in the packages API Proxy and API Service Element.

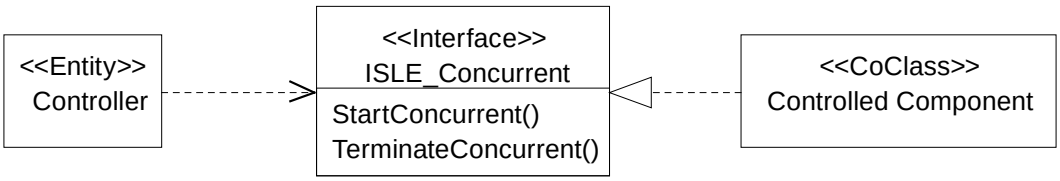

Figure 2-9: Concurrent Control Interface

2.3.4.4 Trace Control Interface

Components supporting diagnostic traces implement the interface `ISLE_TraceControl` to start and stop tracing with a specified trace level. The interface is shown in figure 2-11 in 2.3.5 together with the interface `ISLE_Trace` provided by the application. Specific uses are described in the sections dealing with the API Proxy and the API Service Element.

2.3.4.5 Interfaces Defined by the Package

Name	Description
`ISLE_EventMonitor`	The interface supports registration of external events, for which the event monitor shall wait together with a reference to the interface `ISLE_EventProcessor` to call when the event is detected.
`ISLE_EventProcessor`	The interface provides a method to call when an external event is detected and a method to invoke, if the event monitor aborts.
`ISLE_TimerHandler`	The interface allows starting of a timer together with a reference to the interface `ISLE_TimeoutProcessor` to call when the timer expires. It also provides a method to cancel a running timer.
`ISLE_TimeoutProcessor`	The interface provides a method to call, when a timer expires, and a method to invoke, if the timer handler aborts.
`ISLE_Sequential`	The interface provides methods to start and terminate the operation of a component that can only handle sequential flows of control. It allows passing of interfaces to an event monitor and a timer handler.
`ISLE_Concurrent`	The interface provides methods to start and terminate the operation of a component supporting concurrent flows of control.
`ISLE_TraceControl`	The interface provides methods to start tracing and stop tracing.

2.3.5 PACKAGE SLE APPLICATION

2.3.5.1 Overview

The SLE Application is not an API component, but the client of the API. However, the application must provide a set of interfaces for use by the API. In addition, the application must perform configuration, initialization and control of the API.

The obligations of the application and the interfaces it provides are described in this model by a set of classes that are assumed present in the application program. These classes are pure modeling constructs and do not prescribe the design and implementation of the application in any way.

Figure 2-10 shows the classes provided for actual service provisioning. The model assumes that every service instance is handled by an instance of the class API Application Instance. This class defines the functionality that is independent of the user or provider role and the specific SLE transfer service type. Specific derived classes are assumed for every service type and role. These are represented by the classes <SRV> User Application and <SRV> Provider Application in the figure.

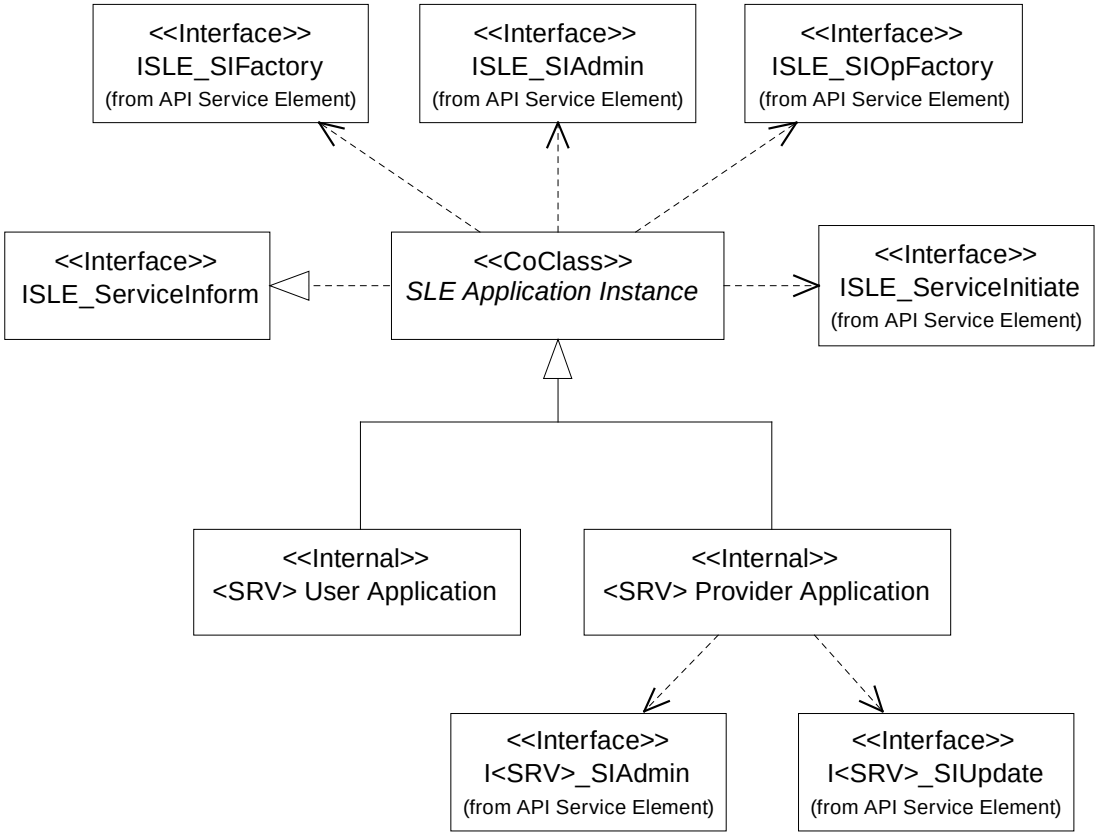

Figure 2-10: Structure of the Package SLE Application

Interfaces that must be provided by the application for logging, for notification of events, and for diagnostic traces, are shown in figure 2-11. The model assumes a class that accepts log records and notifications (Reporter) and a class that accepts trace records (Trace). The model does not make any assumptions about the number of objects an application uses.

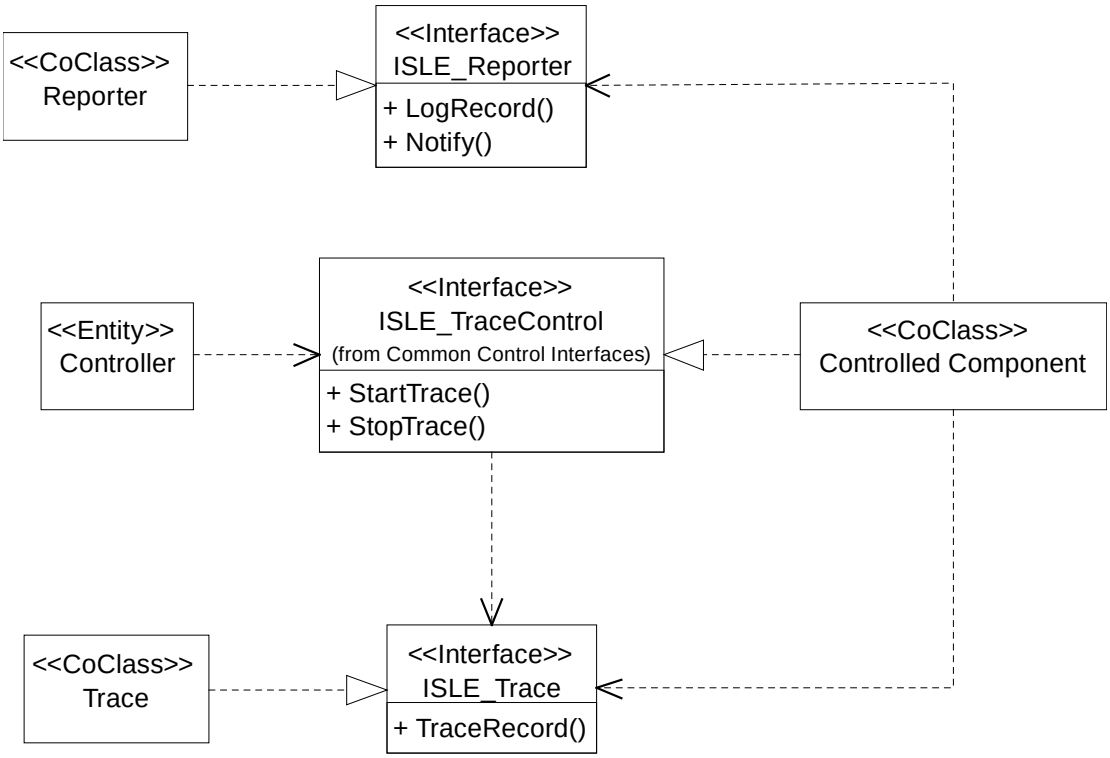

Figure 2-11: Reporting and Tracing Interfaces Provided by the Application

Finally, applications have the option of supplying an external time source to the API components. To use this option, applications must provide an implementation for the interface `ISLE_TimeSource` (Component Class Time Source) and pass it to the creator function of the component SLE Utilities (see 2.3.7). If the interface is supplied by the application, the component uses the interface to retrieve current time. Otherwise, it uses system time.

In addition to the tasks discussed in this section, the application is responsible for configuration, initialization and shutdown of API components. These tasks are discussed in more detail in annex F.

2.3.5.2 Component Class SLE Application Instance

2.3.5.2.1 General

The component class SLE Application Instance handles a single service instance. For this purpose it implements and exports the interface `ISLE_ServiceInform` by which it receives SLE PDUs sent by the peer SLE application. This interface is identical for all service types and all roles of an SLE Application. The class SLE Application Instance uses the interface `ISLE_ServiceInitiate` to pass PDUs to the service instance in the component API Service Element.

The application instance creates the service instance in the service element using the interface `ISLE_SIFactory` and configures the service instance using the interface `ISLE_SIAdmin`. For invocations of SLE operations, the class uses the interface `ISLE_SIOpFactory` to create the required operation objects.

2.3.5.2.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_ServiceInform`	SLE Application	passing of SLE PDUs to the application

2.3.5.2.3 Dependencies

Interface	Defined in Package	Purpose
`ISLE_SIFactory`	API Service Element	creation and deletion of service instances
`ISLE_SIAdmin`	API Service Element	configuration of the service instance
`ISLE_SIOpFactory`	API Service Element	creation and initialization of operation objects
`ISLE_ServiceInitiate`	API Service Element	passing of SLE PDUs from the application to the service instance

2.3.5.3 Internal Class <SRV> User Application

The class <SRV> User Application represents a set of specific classes handling service instances of a specific service type for an SLE user application.

2.3.5.4 Internal Class <SRV> Provider Application

2.3.5.4.1 General

The class <SRV> Provider Application represents a set of specific classes handling service instances of a specific service type for an SLE provider application. The class must set service-specific configuration parameters in the service instance of the service element

component. If specified by the relevant supplemental Recommended Practice for the service-specific API, it updates the service parameters of the service instance using the interface `I<SRV>_SIUpdate`.

2.3.5.4.2 Dependencies

Interface	Defined in Package	Purpose
`I<SRV>_SIAdmin`	Service Supplement	configuration of the service instance
`I<SRV>_SIUpdate`	Service Supplement	update of service parameters

2.3.5.5 Component Class Reporter

2.3.5.5.1 General

The component class Reporter implements the interface `ISLE_Reporter`, by which the application receives log messages and notifications. It is assumed that the log messages are stored to the system log and notifications are brought to the attention of the operator. A reference to the interface is passed to the API Proxy and the API Service Element when they are configured.

2.3.5.5.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_Reporter`	SLE Application	logging and notification

2.3.5.6 Component Class Trace

2.3.5.6.1 General

The component class Trace implements the interface `ISLE_Trace`, by which the application receives trace records. It is assumed that the class stores the trace records to a file. A reference to the interface is passed to the tracing component with the method `StartTrace()` in the interface `ISLE_TraceControl`.

2.3.5.6.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_Trace`	SLE Application	Tracing

2.3.5.7 Component Class Time Source

2.3.5.7.1 General

The component class Time Source implements the interface `ISLE_TimeSource`, by which the component class Time (see 2.3.7.3) can retrieve current time. As all API components are obliged to use the interface `ISLE_Time`, the time reference supplied by `ISLE_TimeSource` is distributed throughout the API.

The time provided via the interface `ISLE_TimeSource` can be offset from the system time. However, API components can rely on the fact that the offset is constant throughout the lifetime of an API instance within the limits of the time accuracy defined by this Recommended Practice.

2.3.5.7.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_TimeSource`	SLE Application	Retrieval of current time

2.3.5.8 Interfaces Defined by the Package

Name	Description
`ISLE_ServiceInform`	The interface provides the methods to pass on to the application SLE operation invocations and returns received from the peer application. In addition, it supports resuming of data transfer if that has been suspended.
`ISLE_Reporter`	The reporter interface provides methods to enter a log record to the system log and to notify the application of events that require immediate attention
`ISLE_Trace`	The tracing interface provides a method to pass a trace record.
`ISLE_TimeSource`	Supply of current time.

2.3.6 PACKAGE SLE OPERATIONS

2.3.6.1 Overview

Operation objects store the invocation and return parameters of an SLE operation and export interfaces by which these parameters can be read and written. In addition, the interfaces provide features to verify completeness and consistency of the parameters.

Operation objects are implemented by a separate component because they must be passed across component boundaries.

An implementation provides one operation object class for every operation defined for the SLE transfer service types it supports. All implementations support the common operations defined in 2.3.6.7 and 2.3.6.8. In addition, all implementations provide an Operation Factory, providing the interface `ISLE_OperationFactory` to create an operation object with a specified interface, a specified SLE transfer service type and a specified version number for the service type.

Common characteristics of all operation objects are defined by the abstract component class Operation and its interface `ISLE_Operation`. Common characteristics of confirmed operations are defined by the abstract component class Confirmed Operation and its interface `ISLE_ConfirmedOperation`. All operation objects for unconfirmed SLE operations are derived from Operation and all operation objects for confirmed SLE operations are derived from Confirmed Operation. The same applies to the interfaces exported by these objects.

NOTE – Classes in the package SLE Operations use the interfaces of utility objects. This fact is not specifically mentioned in the following description.

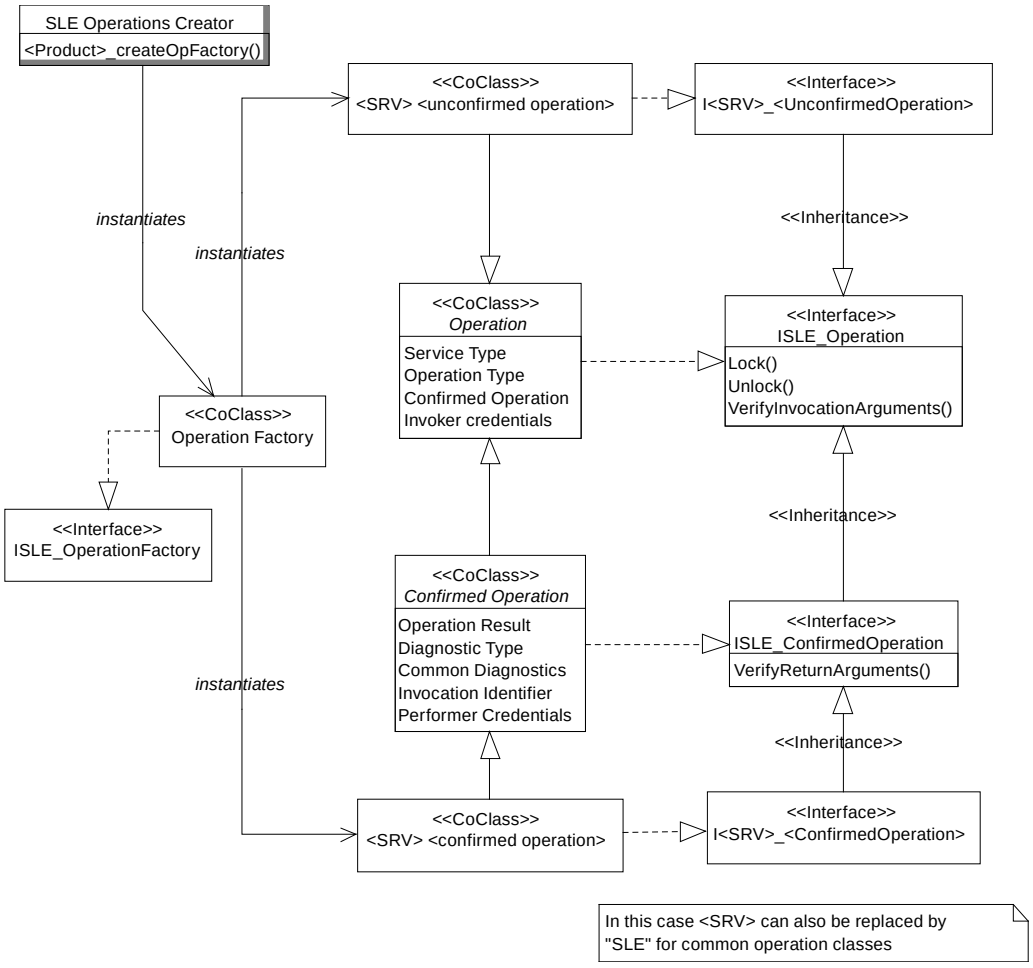

Figure 2-12: Operation Objects

2.3.6.2 Component Class Operation Factory

2.3.6.2.1 General

The operation factory provides an interface to create an instance of an operation object by specification of the desired interface, the operation type, the service type and the version number of the service type.

2.3.6.2.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_OperationFactory`	SLE Operations	creation of operation objects

2.3.6.3 Component Class Operation

2.3.6.3.1 General

The class defines common characteristics supported by all operations objects.

2.3.6.3.2 Attributes

2.3.6.3.2.1 Common Attributes

The common attributes are displayed in figure 2-12. These are accessible via the interface `ISLE_Operation`, inherited by all operation objects.

2.3.6.3.2.2 Service Type and Operation Type

An operation object class is uniquely identified by the combination of the SLE transfer service type and the operation type, because the same operation can have different parameters for different SLE transfer service types.

2.3.6.3.2.3 Version Number

Every operation object identifies the version number of the service it supports, because use of the operation object might differ between the versions.

2.3.6.3.2.4 Confirmed Operation

Identifies whether the operation is confirmed or not.

2.3.6.3.2.5 Invoker Credentials

Holds the credentials of the invoker.

2.3.6.3.3 Behavior and Use

2.3.6.3.3.1 Checking of Invocation Parameters

Interfaces to operation objects provide a method for checking the invocation arguments with respect to completeness, consistency and range. Obviously, an operation object cannot perform checks that require knowledge of the context. The checks performed are defined in A5 for common operations and in the supplemental Recommended Practice documents for service-specific APIs for service type-specific operations.

2.3.6.3.3.2 Support for Concurrent Flows of Control

Access to operation objects is not safe with respect to concurrent access by multiple threads. However, operation objects provide an advisory lock, which can be used to ensure that access to the object is guarded. The guarding mechanism provided by operation objects prevents self inflicting locks.

2.3.6.3.4 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_Operation`	SLE Operations	access to common attributes of operation objects

2.3.6.4 Component Class Confirmed Operation

2.3.6.4.1 General

The class defines common characteristics supported by all operations objects for confirmed SLE operations.

2.3.6.4.2 Attributes

2.3.6.4.2.1 Common Attributes

The common attributes are displayed in figure 2-12.

2.3.6.4.2.2 Operation Result

Operation Result holds the result of the operation when it has been performed.

2.3.6.4.2.3 Diagnostic Type

Diagnostic Type identifies whether diagnostics are present and if so, whether the common diagnostics or special diagnostics have been used.

2.3.6.4.2.4 Common Diagnostics

Common Diagnostics holds the common diagnostics, if present.

2.3.6.4.2.5 Invocation Identifier

The Invocation Identifier holds the invocation identifier defined for SLE services.

2.3.6.4.2.6 Performer Credentials

Performer Credentials holds the credentials of the performer.

2.3.6.4.3 Behavior and Use

2.3.6.4.3.1 Checking of Return Parameters

Interfaces of confirmed operation objects provide a method for checking the return arguments with respect to completeness, consistency and range. The checks performed are defined in A5 for common operations and in the supplemental Recommended Practice documents for service-specific APIs for service type-specific operations.

2.3.6.4.3.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_ConfirmedOperation`	SLE Operations	access to common attributes of confirmed operation objects

2.3.6.5 Component Class <SRV> <Unconfirmed Operation>

An operation object class is provided for every unconfirmed SLE operation of the SLE transfer service types supported by the component. The interfaces of these classes are derived from `ISLE_Operation`. The names of the interfaces are constructed by replacing <SRV> by the abbreviation for the service type. For instance, the name of the interface for the TRANSFER-DATA operation of the RAF service is `IRAF_TransferData`.

2.3.6.6 Component Class <SRV> <Confirmed Operation>

An operation object class is provided for every confirmed SLE operation of the SLE transfer service types supported by the component. The interfaces of these classes are derived from `ISLE_ConfirmedOperation`. The names of the interfaces are constructed by replacing <SRV> by the abbreviation for the service type. For instance, the name of the interface for the TRANSFER-DATA operation of the FSP service is `IFSP_TransferData`.

2.3.6.7 Operations for Common Association Management

2.3.6.7.1 General

The SLE operations for association management are used for all service types. The interfaces of operation objects for common association management are shown in figure 2-13.

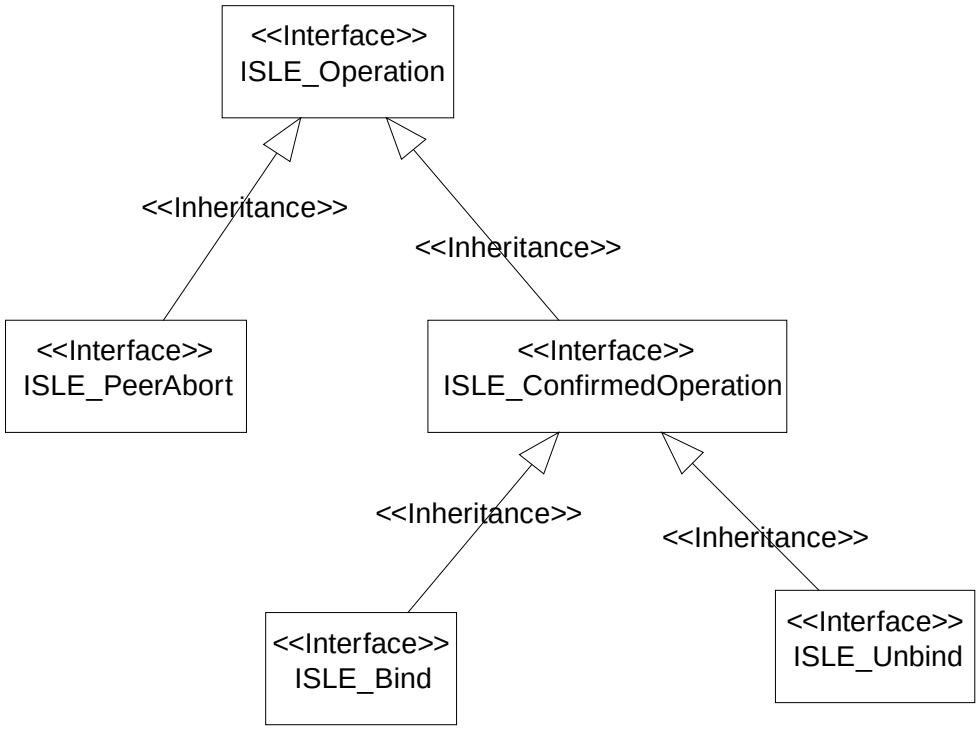

Figure 2-13: Operation Object Interfaces for Common Association Management

2.3.6.7.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_Bind`	SLE Operations	access to parameters of the BIND operation
`ISLE_Unbind`	SLE Operations	access to parameters of the UNBIND operation
`ISLE_PeerAbort`	SLE Operations	access to parameters of the PEER-ABORT operation

2.3.6.8 Other Common SLE Operations

2.3.6.8.1 General

The operations shown in figure 2-14 are identical for all SLE service types that actually use them. Therefore, the operation object interfaces are defined in this Recommended Practice.

The operation TRANSFER-BUFFER is actually not an SLE operation. In the API it is used to transfer the contents of the transfer buffer defined for return services between components. This object also provides methods to facilitate buffering of other operation objects.

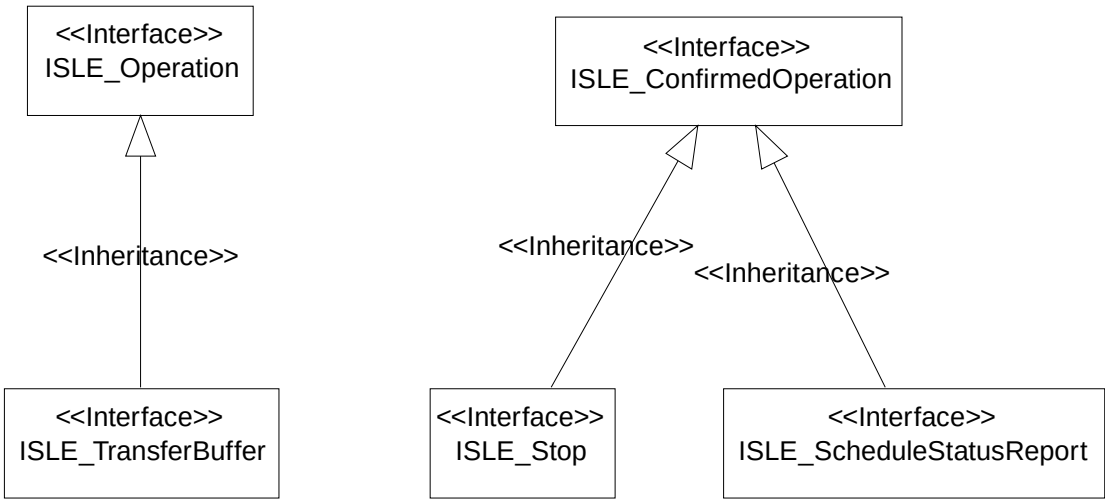

Figure 2-14: Common SLE Operation Objects

2.3.6.8.2 Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_Stop`	SLE Operations	access to parameters of the STOP operation
`ISLE_ScheduleStatusReport`	SLE Operations	access to parameters of the SCHEDULE-STATUS-REPORT operation
`ISLE_TransferBuffer`	SLE Operations	support for handling of the transfer buffer for return services

2.3.6.9 Interfaces Defined by the Package

Name	Description
`ISLE_OperationFactory`	Creation of operation objects
`ISLE_Operation`	Common characteristics of operation objects
`ISLE_ConfirmedOperation`	Common characteristics of confirmed operation objects
`ISLE_Bind`	BIND operation
`ISLE_Unbind`	UNBIND operation
`ISLE_PeerAbort`	PEER-ABORT operation
`ISLE_Stop`	STOP operation
`ISLE_ScheduleStatusReport`	SCHEDULE-STATUS-REPORT operation
`ISLE_TransferBuffer`	Support for handling of the transfer buffer for return services

2.3.7 PACKAGE SLE UTILITIES

2.3.7.1 Overview

The package SLE Utilities defines a small set of utility classes and the associated interfaces. The utilities defined for the API are shown in figure 2-15.

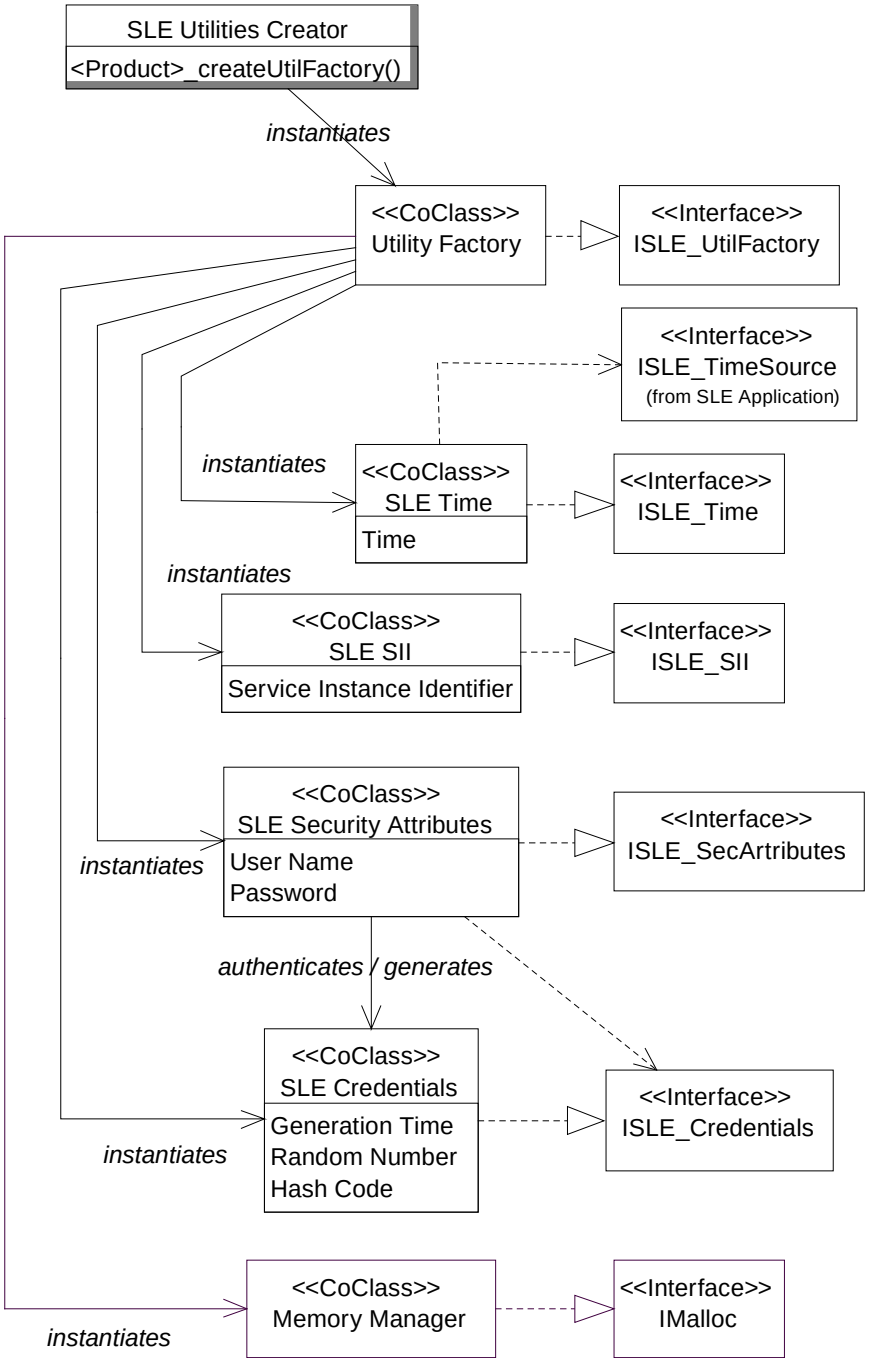

Figure 2-15: SLE Utilities

2.3.7.2 Component Class Utility Factory

The Utility Factory provides an interface to create instances of the utility classes, specified by the identifier of the interface. It returns a pointer to the interface exported by the requested class.

2.3.7.3 Component Class Time

The SLE Time class provides a limited set of time handling functions. It specifically supports the CCSDS defined time codes and conversion between these codes and the native time representation of the platform. Its services are available via the interface `ISLE_Time`.

If an external time source interface (`ISLE_TimeSource`) was supplied to the creator function of the component, the class Time uses that interface to determine current time. Otherwise it uses system time.

2.3.7.4 Component Class Service Instance Identifier

The class handles the service instance identifier defined by the CCSDS Recommended Standards for SLE transfer services. It supports a standard ASCII representation of the service instance identifier (see annex C for version 1 of the SLE services RAF, RCF and CLTU, and references [4], [5] and [7] for version 2 of the SLE services RAF, RCF and CLTU, and [6] and [8] for the SLE services ROCF and FSP), and verifies that the components of the identifier are those defined by CCSDS. Its services are available via the interface `ISLE_SII`.

2.3.7.5 Component Class Credentials

The class holds the credentials used for authentication of the peer identity and provides access to its attributes via the interface `ISLE_Credentials`.

2.3.7.6 Component Class Security Attributes

The class holds the user name and password for generation of credentials and for authentication of the peer identity. It implements generation of the credentials from the attributes stored and authentication of credentials received from a peer application. Its services are available via the interface `ISLE_SecAttributes`.

2.3.7.7 Component Class Memory Manager

The class provides memory management that must be used for all data structures passed across component boundaries and between the application and API components.

2.3.7.8 Interfaces Defined by the Package

Name	Description
`ISLE_UtilFactory`	Creation of SLE utility objects
`ISLE_Time`	Time handling
`ISLE_SII`	Handling of the service instance identifier
`ISLE_Credentials`	Storage and transfer of credentials for authentication
`ISLE_SecAttributes`	Storage of security attributes for authentication, generation of credentials, and authentication of credentials
`IMalloc`	Memory management

2.4 SECURITY ASPECTS OF CORE SLE API CAPABILITIES

The security aspects of the core SLE API capabilities specified in this Recommended Practice are highly dependent upon the specific SLE Transfer Services that use these core API capabilities. Therefore, the security aspects associated with the SLE API are identified as part of the Recommended Practices for each of the specific SLE transfer services, and are not further addressed in this specification.

3 SPECIFICATION OF API COMPONENTS

3.1 INTRODUCTION

This section provides detailed specifications for the API components

- API Proxy (see 3.2);
- API Service Element (see 3.3);
- SLE Operations (see 3.4);
- SLE Utilities (see 3.5).

In addition, 3.6 specifies what the API expects from an SLE application both in terms of interfaces that must be implemented and in terms of the tasks the application is expected to perform for control of the API. The specification defines the full scope of the API, including the API Service Element. When an application chooses to use the API Proxy directly (e.g., in an SLE gateway as outlined in section 2), it must implement the functionality defined for the API Service Element and must export all interfaces needed by API Proxy.

As far as possible, each of the API components and the SLE application are specified by a self-contained subsection. In some cases, these subsections comprise specifications on how interfaces exported by a component must be used. Such specifications actually define requirements on clients of the component. Where this is the case, cross-references have been entered to the subsections dealing with client components. Common specifications applicable to all of the components or on a subset of the components are provided in 3.7.

The specification of API components in this section is based on the model described in section 2. It is complemented by state transition tables for the API Proxy and the API Service Element in section 4, and by the specification of the interfaces in annex A.

3.2 API PROXY

3.2.1 FEATURES

3.2.1.1 The proxy shall implement all aspects of SLE transfer services that need to be provided by technology-specific means.

3.2.1.1.1 The proxy shall perform data conversion between SLE PDUs transmitted across the network and operation objects used within the API, see 3.2.2.

3.2.1.1.2 The proxy shall establish, maintain, and terminate data communication associations with one or more peer proxies, see 3.2.4.

3.2.1.1.3 The proxy shall configure and initialize the data communication service as part of its own configuration and initialization procedure, see 3.2.12.

3.2.1.1.4 The proxy shall provide means to dynamically register and de-register responder ports for SLE service provisioning, see 3.2.5.

3.2.1.2 The proxy shall queue protocol data units and provide flow control features both for the data received from the network and data sent to the network, see 3.2.3.

3.2.1.3 The proxy shall implement access control on system level, perform authentication of the peer identity for SLE PDUs received from the network, and generate the credentials for PDUs sent to the network, see 3.2.6.

3.2.1.4 The proxy shall generate entries to the log of the hosting system for important events, see 3.2.8.

3.2.1.5 The proxy shall provide the feature to produce an event trace for the complete proxy and for individual associations, see 3.2.9.

3.2.1.6 The proxy shall support a range of execution environments with respect to use of processes and in process threads, see 3.2.10.

3.2.1.7 The proxy shall use a configuration database, which shall control its operation within a specific deployment environment, see 3.2.11.

3.2.1.8 The proxy shall support a special 'pass-through' mode of operation, in which it does not modify any parameters in the PDUs but forwards them unmodified to the respective recipient.

NOTE – The pass-through mode of operation is further detailed in 3.2.7. Unless stated otherwise, all other specifications refer to the default mode of operation.

3.2.2 PROCESSING OF SLE PROTOCOL DATA UNITS

NOTE – The proxy may support more than one concurrent bound association (see E5.1). In that case the processing specified in this subsection must me performed independently for each association.

3.2.2.1 The proxy shall accept operation objects provided by the component SLE Operations via the interface `ISLE_SrvProxyInitiate`. With the operation parameters extracted from the operation objects, it shall create SLE protocol data units in the format and encoding required for transfer, and transmit these PDUs to the peer proxy.

NOTE – The format and encoding used for transfer are determined by the technology used by the proxy.

3.2.2.1.1 For invocations of unconfirmed SLE operations and for operation returns, the proxy shall release the operation object when the parameters have been extracted from the object.

NOTE – The exact time at which the object is released is not defined by this specification. The objects might be released immediately or when the PDU has been actually transmitted.

3.2.2.1.2 For invocations of confirmed operations, the proxy shall memorize the object until the operation return from the peer proxy arrives, or the association is terminated.

3.2.2.1.3 Except for the PEER-ABORT invocation, the proxy shall ensure that PDUs received from its local client on one association are transmitted to the peer proxy in the sequence received.

NOTE – Handling of the PEER-ABORT invocation is defined in 3.2.4.4.

3.2.2.2 The proxy shall receive SLE protocol data units from the peer proxy in the format and encoding used for transfer. It shall decode the operation parameters, store them to the associated operation object and forward the operation object to its client via the interface `ISLE_SrvProxyInform`.

3.2.2.2.1 For operation invocations, the proxy shall create a new operation object, using the interface `ISLE_OperationFactory`, which is supplied to the proxy as part of its configuration.

3.2.2.2.2 For operation returns, the proxy shall associate the PDU received from the peer proxy with the operation object of the corresponding invocation using the invocation identifier. If no corresponding invocation can be found, the proxy shall abort the association with PEER-ABORT and the diagnostic ‘unsolicited invoke ID’.

NOTES

1 The invocation identifier of a return must match the invocation identifier of an invocation for the same operation type. It is noted that invocation identifiers must also be unique across all operations. This requirement is handled by the service element.

2 It is further noted that the confirmed operations BIND and UNBIND do not carry an invocation identifier. For these operations, only a single return can be outstanding at any time, such that association of the return with the invocation shall be possible.

3.2.2.2.3 The proxy shall release an operation object passed to its client when the function passing that object has returned.

NOTE – The specification implies that the proxy does not memorize invocations of confirmed operations it passes to its local client. It is considered the responsibility of the client not to send any returns for which no invocation has been received.

3.2.2.2.4 If there is a decoding error, the proxy shall abort the association using the PEER-ABORT operation.

3.2.2.2.5 Except for the PEER-ABORT invocation, the proxy shall ensure that PDUs received from the peer proxy on one association are delivered to its client in the sequence received.

NOTE – Handling of the PEER-ABORT invocation is defined in 3.2.4.4.

3.2.3 FLOW CONTROL

NOTE – The proxy may support more than one concurrent bound association (see E5.1). In that case the processing specified in this subsection must me performed independently for each association.

3.2.3.1 Incoming Traffic

The proxy shall limit the number of protocol data units received from the peer proxy and not yet forwarded to its client to a configurable maximum number N1 per association. In addition, the proxy shall ensure that a maximum number N2 ≤ N1 of these are TRANSFER-BUFFER invocations. When either of these limits is reached the proxy shall not read any further data from the network such that a backlog is built up.

NOTES

1 The objective of this specification is to ensure that incoming traffic is controlled and backpressure is actually built up when needed. An implementation may restrict the number of PDUs it accepts from the network per association or for all associations; it must not accept more than defined by the configuration parameters. When a proxy limits the number of incoming PDUs that it can process in parallel, the limits shall be clearly documented.

2 TRANSFER-BUFFER is not defined as an SLE operation but refers to the PDU used for transmission of the transfer buffer used by return link services. For a specification of the ‘pseudo-operation’ TRANSFER-BUFFER within the API, see 3.4.

3.2.3.2 Outgoing Traffic

3.2.3.2.1 The proxy shall queue a configurable maximum number of PDUs for transfer per association.

NOTE – This specification does not prescribe what the proxy actually queues. Whether it queues the operation object, a data structure ready for transmission, or any other object depends on the implementation. For a technology based on remote procedure calls, the 'queue' might also consist of procedure calls that have not yet been completed.

3.2.3.2.2 When the proxy accepts a transfer request by a result code indicating success, it shall guarantee that the associated PDU has been queued for transmission. The positive result code does not imply that the PDU has been transmitted.

3.2.3.2.3 When the client requests to be informed on transmission of a PDU (setting the argument `reportTransmission` in the interface `ISLE_SrvProxyInitiate` to `true`), the proxy shall inform the client by:

a) returning the appropriate result code of the function if the PDU can be sent immediately; or

b) calling the function `PDUTransmitted()` in the interface `ISLE_SrvProxyInform` when that PDU has been transmitted if immediate transfer is not possible.

NOTE – The exact meaning of 'transmitted' depends on the technology used by the proxy. As a minimum, the communications system must have been requested to initiate transfer of the data.

3.2.3.2.4 When the maximum queue size has been reached, the proxy shall reject further transfer requests with a result code indicating 'overflow' until the queue size drops below the threshold again.

NOTE – It is expected that the client will abort the association in such a case. However this decision must be taken by the client and not by the proxy.

3.2.3.2.5 The proxy shall provide a method to discard TRANSFER-BUFFER invocations that have been queued for transmission and for which data transfer has not yet started.

NOTE – In other respects, the proxy shall handle the TRANSFER-BUFFER invocation as any other PDU. It shall queue more than one TRANSFER-BUFFER invocation if so requested. This is necessary in some cases, e.g., for support of 3.3.5.3.5.1 item b). Buffering for return services is the responsibility of the service instance.

3.2.3.2.6 When discarding of queued buffers is requested (using the method `DiscardBuffer()` in the interface `ISLE_SrvProxyInitiate`), the proxy shall search the queue and release all resources allocated for all TRANSFER-BUFFER PDUs on the queue.

3.2.3.2.7 The result code returned by the method shall indicate whether one or more TRANSFER-BUFFER PDUs have been actually discarded.

3.2.4 ASSOCIATION MANAGEMENT

3.2.4.1 General Specifications

NOTES

1 The exact meaning of an association depends on the technology used by the proxy. In the context of the SLE API, the essentials are:

a) an association is established when the BIND operation has been completed successfully;

b) an association is terminated by one of the operations UNBIND, PEER-ABORT, or by a protocol abort;

c) other SLE operation invocations and returns can only be exchanged on an established association.

2 This specification makes no assumptions concerning the characteristics of the technology and its use by the proxy.

3 If the underlying technology is connection oriented, an implementation might:

a) apply a one to one mapping between an association and a connection;

b) use multiplexing of associations on one connection; or

c) use more than one connection for a single association.

4 This specification does not prescribe whether the operations BIND, UNBIND and PEER-ABORT are implemented by means of specific connection establishment and release procedures provided by the communications technology or by exchange of data on an established connection.

5 If the technology is connectionless, the notion of an association is provided by the implementation of the proxy.

6 In this specification, the term association is also used to refer to the component object that provides the interface to the ‘real association’. The association object can exist in an ‘unbound’ state; i.e., the association it handles has not yet been established or has been terminated. Whether a specification refers to the association object or the actual association should become clear from the context in most cases. Where there is a need to explicitly refer to the object, the term ‘association object’ is used. To make explicit reference to the association provided by the data communications service, the term ‘data communication association’ is used.

7 Further details are specified in the state table for associations in section 4. This state table complements the following specifications.

3.2.4.1.1 The proxy shall establish a data communication association with a peer proxy as part of the BIND operation.

NOTE – Association establishment by the proxy is specified in 3.2.4.2.

3.2.4.1.2 Associations managed by a proxy are distinguished by the role the proxy plays in the BIND operation. The role of an association can be either 'initiator' or 'responder'.

NOTE – The roles 'initiator' and 'responder' for the BIND operation are defined in references [4], [5], [6], [7] and [8].

3.2.4.1.3 An implementation of the API Proxy may support associations in the initiator role and associations in the responder role concurrently or may provide associations only for one of these roles.

3.2.4.1.4 Associations in the initiator role shall be created and deleted by the client using the interface `ISLE_AssocFactory` exported by the proxy.

3.2.4.1.5 If the implementation does not support associations in the initiator role or does not support the SLE service type requested by the client, the association factory shall reject the request.

3.2.4.1.6 Following creation of an association in the initiator role, its state shall be 'unbound'.

3.2.4.1.7 Associations in the initiator role shall use the interface `ISLE_SrvProxyInform` passed to the factory interface to forward SLE PDUs received from the peer proxy.

3.2.4.1.8 The proxy shall release association objects in the initiator role only on request of the client (via the interface `ISLE_AssocFactory`) or as part of the terminate function. It shall reject the request to delete the association object if the association is not in the state 'unbound'.

3.2.4.1.9 Association objects in the responder role shall be created and deleted autonomously by the proxy as part of the association establishment and release procedures.

NOTE – Association establishment and release is specified in 3.2.4.2, 3.2.4.3 and 3.2.4.4.

3.2.4.1.10 The proxy shall provide specific associations for every SLE service type it supports.

3.2.4.1.11 An association shall accept every PDU defined for the supported SLE service from its local client or the remote proxy. For any other PDU, the proxy shall reject a transfer request from its local client and abort the association with PEER-ABORT, if it receives the PDU from the peer proxy.

3.2.4.1.12 The proxy shall be able to decode PDUs received on an association if the PDUs are defined for the service type supported by the association object.

NOTE – For other PDUs, decoding is expected to fail. If decoding does succeed, the PDU must be rejected as 'unknown' according to 3.2.4.1.11.

3.2.4.1.13 The proxy and the associations shall not distinguish between the roles SLE service provider and SLE service user. Associations shall accept every PDU that is defined for the supported SLE service type from the local client and from the remote proxy.

NOTE – This implies, for instance, that an association might also accept a TRANSFER-DATA invocation for a forward service when that is issued by an SLE service provider. It is the responsibility of higher layers to prevent such requests. Note that associations do distinguish between the role 'initiator' and 'responder', and apply the associated rules for the BIND and UNBIND operations defined in 3.2.4.

3.2.4.1.14 The proxy shall terminate a data communication association in an orderly manner as part of the UNBIND procedure.

NOTE – Orderly association release by the proxy is specified in 3.2.4.3.

3.2.4.1.15 The proxy shall abort an association in the following cases:

a) the local client invokes the PEER-ABORT operation;

b) the remote proxy invokes the PEER-ABORT operation;

c) abort of the association is explicitly required by any other specification for the proxy in this document; or

d) the proxy is affected by major problems and cannot continue processing of the association.

NOTE – This specification implies that the proxy might also abort the association in the case of a catastrophic failure when that case is not specified in this document. Association abort in is specified in 3.2.4.4.

3.2.4.2 Association Establishment

NOTE – This section defines procedures for association establishment without consideration of security aspects. Specifications related to access control and authentication, which must be taken into account for association establishment, are provided in 3.2.6.

3.2.4.2.1 Associations in the Initiator Role

3.2.4.2.1.1 For association objects in the initiator role, the proxy shall initiate establishment of a data communication association when the client requests transfer of a BIND invocation and the state of the association object is ‘unbound’.

3.2.4.2.1.2 The proxy shall initiate association establishment using the parameters of the BIND invocation. It shall transmit the BIND invocation PDU to the remote proxy as part of this procedure.

3.2.4.2.1.3 The proxy shall complete the association establishment procedure when it receives the BIND return from the remote proxy. If the BIND return PDU contains a positive result, the association shall be established and the state shall be set to ‘bound’. If the BIND return PDU carries a negative result, the association shall not be established and the state shall be set to ‘unbound’. The proxy shall inform its client by forwarding the operation object with the return parameters received from the peer proxy.

3.2.4.2.1.4 If association establishment fails before the BIND invocation can be transmitted or before the BIND return is received, the proxy shall inform its client and perform the cleanup actions defined for the PEER-ABORT operation in 3.2.4.4.

NOTE – This specification does not prescribe the means by which the proxy informs its client, as the selection of the appropriate method depends on implementation details. If the connection failure is detected in the same thread of control in which the BIND invocation was passed to the proxy, the proxy may opt to return the appropriate error code to the caller of that method. In all other cases, the proxy shall use the method `ProtocolAbort()` of the interface `ISLE_SrvProxyInform`.

3.2.4.2.2 Associations in the Responder Role

3.2.4.2.2.1 A proxy supporting associations in the responder role may listen for association establishment requests on the network interface using technology-specific means. Whether a proxy instance actually listens for such requests and when it starts listening is defined in the configuration database.

NOTE – In a given deployment environment, a proxy may not be supposed to listen for and respond to BIND invocations from the network interface, although it may be able to do so. An example of such an environment is an SLE service user system that does not support the provider-initiated bind option. The configuration database and the initialization procedures are specified in 3.2.11. Depending on the technology used, the implementation of the proxy, and the requirements of the hosting system, a proxy might only start listening when a port has been registered dynamically. In other environments, the port on which a proxy listens might be statically defined and the proxy might start listening as soon as its operation has been started.

3.2.4.2.2.2 A proxy listening for association establishment requests on the network interface shall process an incoming call as defined by the specifications in 3.2.4.2.2.3 to 3.2.4.2.2.10.

3.2.4.2.2.3 The proxy shall receive a BIND invocation PDU from the remote proxy as part of the association establishment procedure, and shall perform the checks defined in 3.2.4.2.2.4 and 3.2.4.2.2.5.

NOTE – Further checks related to access control and authentication are defined in 3.2.6.

3.2.4.2.2.4 If the SLE service type does not match one entry in the list of supported service types in the configuration database, the proxy shall respond with a BIND return containing a negative result and the diagnostic ‘service type not supported’.

3.2.4.2.2.5 If the version number does not match one entry in the list of supported versions for the service type defined in the configuration database, the proxy shall respond with a BIND return containing a negative result and the diagnostic ‘version not supported’.

NOTE – This version of the API does not support the optional version-number negotiation procedure defined by the CCSDS Recommended Standards for SLE transfer services. The responding proxy either accepts the proposed version number, or responds with a BIND return containing a negative result. It does not propose a different version number.

3.2.4.2.2.6 If the BIND invocation is acceptable for the proxy, it shall create an association object supporting the SLE service type identified in the BIND invocation. It shall then inform its client using the interface `ISLE_Locator` and pass a reference to interface `ISLE_SrvProxyInitiate` of the association as well as to the BIND operation object.

3.2.4.2.2.7 If the locator returns a positive result code and a pointer to the complementary interface `ISLE_SrvProxyInform`, the proxy shall forward the BIND operation object via that interface.

3.2.4.2.2.8 If the locator returns an error, the proxy shall send a BIND return containing a negative result and a diagnostic reflecting the result code returned by the locator to the remote proxy. The proxy shall not establish the data communication association and shall release the association object.

3.2.4.2.2.9 The proxy shall complete the association establishment procedure when it receives the BIND return from its local client. If the BIND return PDU contains a positive result, the association shall be established and the state shall be set to ‘bound’. If the BIND return PDU carries a negative result, the association shall not be established and the association object shall be released. In both cases, the proxy shall forward the BIND return to the peer proxy.

3.2.4.2.2.10 If association establishment fails before the call to the locator, the proxy shall release all resources allocated to the association and shall not inform its client. If association establishment fails subsequently but before the BIND return can be transmitted, the proxy shall inform the client and perform the cleanup actions defined for the PEER-ABORT operation in 3.2.4.4.

3.2.4.2.3 Port Identifiers

3.2.4.2.3.1 The proxy shall map the responder port identifier specified by the CCSDS Recommended Standards for SLE transfer services to address information as required by the technology used.

NOTE – The means by which this mapping is performed is not prescribed by this specification. Options include a local table lookup and a query to a directory system. The method used by an implementation must be documented together with the required configuration.

3.2.4.2.3.2 When a BIND invocation is requested on the local interface, the proxy shall derive the technology dependent information required to establish an association from the parameter 'responder port identifier'.

3.2.4.2.3.3 When receiving a BIND invocation from a peer proxy, the proxy shall ensure that the value of the responder port identifier passed to the local client is identical to the value that has been passed to the peer proxy by the client of the peer proxy.

NOTE – An implementation may choose to transmit the original value or to derive it from technology-specific formats.

3.2.4.2.4 Protocol for the BIND Operation

The proxy shall ensure that the BIND operation is not performed on an established association or during association release and is not re-invoked during association establishment. It shall also ensure that the BIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

3.2.4.3 Orderly Association Release

3.2.4.3.1 The proxy shall enforce the rules defined in the CCSDS Recommended Standards for SLE transfer services for initiating the UNBIND operation. It shall ensure that the UNBIND operation is performed only on an established association and is not re-invoked during association release. It shall also ensure that UNBIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

3.2.4.3.1.1 When receiving a valid UNBIND invocation from the peer proxy, the proxy shall remove and discard all operation invocations that are queued for transmission on the affected association.

NOTE – Following reception of an UNBIND invocation, the responder shall not send any further invocations. Pending operation returns may still be transmitted.

3.2.4.3.1.2 The proxy shall ensure that the data communication association is terminated as part of the UNBIND operation.

NOTE – The means by which the data communication association is terminated and the time at which it is terminated depends on the technology used and the implementation of the proxy.

3.2.4.3.1.3 Following completion of the UNBIND operation, the proxy shall set the state of the association to 'unbound', release all resources allocated to the association, discard all PDUs that may still be queued for transmission, and release all operation objects that may still be memorized. If the association object has the responder role, the proxy shall also release the association object.

3.2.4.4 Association Abort

3.2.4.4.1 The proxy shall implement invocation of the PEER-ABORT operation as defined by the following specifications.

3.2.4.4.1.1 The proxy shall discard all PDUs that are queued for transmission and all PDUs received from the peer proxy that have not yet been forwarded to its client. It shall also release all operation objects for which returns are still pending.

3.2.4.4.1.2 The proxy shall make sure that the peer proxy recognizes the PEER-ABORT invocation and that the diagnostic parameter of the PEER-ABORT invocation is made known to the peer proxy.

3.2.4.4.1.3 The proxy shall abruptly terminate the data communication association using the most efficient means available from the data communications technology, which are able to meet the requirement stated in 3.2.4.4.1.2.

3.2.4.4.1.4 The proxy shall set the state of the association to 'unbound'. If the association object has the responder role, the proxy shall also release the association object.

3.2.4.4.1.5 If the proxy invokes the PEER-ABORT operation on its own initiative, it shall also forward a PEER-ABORT operation object to its local client. It shall set the parameter 'originator' in this object to 'proxy'.

3.2.4.4.2 When the proxy is informed of a PEER-ABORT invoked by the peer proxy, it shall perform the following steps:

3.2.4.4.2.1 The proxy shall discard all PDUs that are queued for transmission and all PDUs received from the peer proxy that have not yet been forwarded to its client. It shall also release all operation objects for which returns are still pending.

3.2.4.4.2.2 The proxy shall forward a PEER-ABORT operation object containing the diagnostics set by the peer proxy to its local client. It shall set the parameter 'originator' in this object to 'peer'.

3.2.4.4.2.3 The proxy shall set the state of the association to 'unbound'. If the association object has the responder role, the proxy shall also release the association object.

NOTE – Depending on the communications technology and its specific use by the proxy, the proxy might have to accept and discard data that have already been transmitted by the peer proxy. Associated activities are performed 'behind the scenes' and are not visible to clients.

3.2.4.4.3 Aborting an association shall not affect any other associations that are currently handled by the proxy.

3.2.4.5 Failure of the Data Communication Service

3.2.4.5.1 The proxy shall monitor the status of an association and inform its local client if the data communication connection breaks down using the method `ProtocolAbort()` of the interface `ISLE_SrvProxyInform`.

3.2.4.5.1.1 If the communications provider does not signal breakdown of the data communication connection, the proxy shall support the requirement by implementing adequate methods.

3.2.4.5.1.2 The maximum acceptable delay between the time the communications connection fails and the associated call to `ProtocolAbort()` shall be configurable.

3.2.4.5.1.3 If the data communication connection breaks down, the proxy shall perform the cleanup actions defined for the PEER-ABORT operation in 3.2.4.4.

3.2.5 DYNAMIC PORT REGISTRATION

3.2.5.1 The proxy shall support dynamic registration and de-registration of ports on which it accepts BIND invocations, according to the following specifications.

NOTE – The actions associated with dynamic port registration depend on the technology. These could include registration of an address, export of information to a directory service or publishing of a service by any other means. Port registration is only required for a proxy in the role of a BIND responder. If no actions are required for a given technology or implementation, the request should simply be ignored.

3.2.5.1.1 Port registration has the effect that requests sent to the port are correctly routed to the proxy that registered it. If the proxy can detect duplicate registration of the same port by more than one proxy instance, it shall reject the request with an error.

3.2.5.1.2 The information supplied for registration includes the following parameters:

a) responder port identifier; and

b) service instance identifier.

NOTE – The information actually used by the proxy depends on the implementation.

3.2.5.1.3 The proxy shall reject registration if:

a) the proxy does not support associations in the responder role or the responder role is not enabled in the configuration database;

b) the responder port identifier is not part of the address mapping information in the configuration database; or

c) the responder port identifier is not marked as local port.

3.2.5.1.4 If registration is accepted, the proxy shall return a registration identifier that must be used for later de-registration.

3.2.5.1.5 The proxy shall de-register ports on client request as required by the technology.

3.2.5.1.6 The proxy shall ensure that BIND invocations sent to a registered port are received by the proxy instance in the period from port registration until de-registration.

3.2.6 SECURITY

3.2.6.1 Security Information

3.2.6.1.1 The configuration database of the proxy shall contain all information required for authentication of the identity of the local SLE application and of peer SLE applications.

3.2.6.1.1.1 The configuration database shall contain a list of registered peer applications, and for each application:

a) the identifier of the application as a printable character string;

NOTE – The identifier is the name of the authority operating the application (user name).

b) the authentication mode for communication with the peer application; the authentication mode defines one of the following:

1) authentication shall not be applied (‘none’);

2) authentication shall be applied for the BIND operation only (‘bind’); or

3) authentication shall be applied for all operations except for PEER-ABORT (‘all’).

c) the security attributes needed for authentication of the peer application, unless the authentication mode is set to ‘none’.

NOTE – The security attributes are defined in 3.5.6 for the utility object Security Attributes, which provides the actual authentication service.

3.2.6.1.1.2 For the local SLE application, the configuration database shall contain:

a) the identifier of the application as a printable character string; and

b) the security attributes needed for authentication.

3.2.6.1.1.3 Entry and update of the security information in the configuration database is an implementation-specific maintenance activity and is not defined by this specification. An implementation must document the format of the related entries and the means by which the content can be defined and updated.

NOTE – This specification implies that it might be necessary to stop and restart the proxy when this information is modified.

3.2.6.2 Access Control

3.2.6.2.1 When receiving a BIND invocation from the peer proxy for a new association, the proxy shall perform the following steps:

3.2.6.2.1.1 The proxy shall use the parameter ‘initiator identifier’ in the BIND invocation to locate the initiator in the list of registered peer applications.

3.2.6.2.1.2 If the initiator is not registered, the proxy shall respond with a BIND return, containing a negative result and the diagnostic ‘access denied’. This BIND return shall not contain credentials. In addition, the proxy shall generate an ‘access violation alarm’.

NOTE – The access violation alarm is defined in 3.2.6.4.

3.2.6.2.1.3 If the initiator is registered, the proxy shall assign the authentication mode and, if applicable, the security attributes specified for the initiator to the association.

3.2.6.2.2 When receiving a BIND return from its local client, or when generating a BIND return, the proxy shall insert the local application identifier stored in its configuration database into the parameter ‘responder identifier’ of the BIND return PDU.

3.2.6.2.3 When receiving a BIND invocation from its local client for an unbound association, the proxy shall perform the following steps:

NOTE – This specification implies that the authentication mode of an association is always determined by the authentication mode of the peer application. Therefore, different authentication modes may be specified for different peers, but the same authentication mode is always used for the same peer.

3.2.6.2.3.1 The proxy shall use the parameter 'responder identifier' in the operation object passed with the BIND invocation request to locate the responder in the list of registered peer applications.

NOTE – Although the BIND invocation PDU defined by the CCSDS Recommended Standards for SLE transfer services does not include the responder identifier, this argument must be passed to the proxy with the BIND invocation request, to enable the proxy to determine the authentication mode. The proxy does not insert that argument into the BIND invocation PDU.

3.2.6.2.3.2 If the responder is not registered, the proxy shall reject the request with a result code indicating 'peer application not registered'.

3.2.6.2.3.3 If the responder is registered, the proxy shall assign the authentication mode and, if applicable, the security attributes specified for the responder, to the association.

3.2.6.2.3.4 The proxy shall insert the local application identifier stored in its configuration database into the parameter 'initiator identifier' of the operation object and the BIND invocation PDU.

3.2.6.2.4 When receiving a BIND return from the peer proxy on an association in the state 'bind pending', the proxy shall perform the following steps:

3.2.6.2.4.1 If the responder identifier in the PDU is not registered, the proxy shall abort the association with the diagnostic 'access denied' and generate an access violation alarm.

3.2.6.2.4.2 If the responder is registered, but differs from the responder assigned to the association, the proxy shall abort the association with the diagnostic 'unexpected responder ID' and generate an access violation alarm.

3.2.6.3 Authentication

NOTE – In the following, the term 'ignore the PDU' might be interpreted in different ways, depending on the data communications technology, the proxy implementation, and the specific SLE operation. 3.2.6.3.2 specifies permissible interpretations and behaviors.

3.2.6.3.1 The proxy shall perform the following steps for all SLE PDUs received from the peer proxy, immediately after decoding and before any other processing steps.

NOTES

1 For a BIND invocation, these steps are performed immediately after location of the initiator identifier on the list of registered applications, if the PDU is received on an unbound association. For a BIND return, these steps are performed immediately after location of the responder identifier if the PDU is received on an association in the state 'bind pending'.

2 The TRANSFER-BUFFER PDU (see 3.3.5.3) does not represent an invocation or return of an SLE operation but is rather used to transmit a number of TRANSFER-DATA and SYNC-NOTIFY invocations as a single data unit across the association between the SLE Provider and the SLE User. Therefore authentication shall be performed on each of the contained SLE PDUs presenting SLE operation invocations and not on the TRANSFER-BUFFER PDU.

3.2.6.3.1.1 The proxy shall check the authentication mode assigned to the association to determine whether authentication is required.

3.2.6.3.1.2 If authentication is required, the proxy shall use the PDU parameter holding the peer's credentials and the security attributes assigned to the association to authenticate the identity of the peer.

NOTE – For invocations the parameter used is 'initiator credentials' and for returns 'responder credentials'.

3.2.6.3.1.3 For the actual authentication procedure, the proxy shall use the service provided by the component 'SLE Utilities' via the interface `ISLE_SecAttributes`. The acceptable delay argument required for authentication is defined in the configuration database of the proxy.

NOTE – The authentication procedure and the role of the argument 'acceptable delay' are defined in 3.5.6.

3.2.6.3.1.4 If authentication fails, the proxy shall generate an authentication alarm, and ignore the PDU.

NOTE – The authentication alarm is defined in 3.2.6.4.

3.2.6.3.2 The action to 'ignore a PDU' shall be implemented according to the following specifications:

3.2.6.3.2.1 As a rule, the proxy shall not take any action that could be observed via the network. In addition, it shall not modify the state of the association or of any operation object waiting for a return PDU, such that a subsequent 'legal' return will succeed.

3.2.6.3.2.2 In order to prevent permanent blocking of resources, a proxy implementation may abort the underlying data communication connection and set the state of the association

to ‘unbound’ when authentication fails for a BIND invocation, BIND return, or UNBIND invocation. If this option is selected, the abort procedure shall restrict the information that is made available to the peer system to the minimum possible. For a BIND return and an UNBIND invocation, the proxy shall inform its local client using a PEER-ABORT operation object with the diagnostic ‘other reason’.

3.2.6.3.2.3 If an implementation uses technology-specific connection termination procedures for the operation UNBIND, it might not be possible to apply the rule defined in 3.2.6.3.2.1 for a BIND return. In such a case the proxy shall accept the PDU and perform the appropriate actions defined in 3.2.4.3.

NOTE – The proxy shall nevertheless perform authentication and generate the authentication alarm record, such that the attack can be recognized. It is noted that authentication at application level cannot provide any protection against an intruder who succeeds in closing the connection of the underlying communications service. The case addressed here therefore does not imply a reduced level of security.

3.2.6.3.3 When receiving a PDU for transfer from its local client, the proxy shall perform the following steps:

3.2.6.3.3.1 If the authentication mode assigned to the association requires authentication for the PDU, the proxy shall insert the credentials for the local SLE application into the operation object passed by the client.

NOTE – Inserting the credentials into the operation object instead of writing it directly into the PDU makes sure that the information held by the operation object is complete. The credentials are subsequently inserted into the PDU used for transfer.

3.2.6.3.3.2 For generation of the credentials, the proxy shall use the service provided by the component ‘SLE Utilities’ via the interface `ISLE_SecAttributes`.

3.2.6.3.3.3 If authentication is not required, the proxy shall set the parameter for the credentials of the local SLE application in the operation object to ‘not used’.

3.2.6.4 Security Alarms

3.2.6.4.1 For the following security alarms, the proxy shall enter an alarm record into the system log and notify the application via the interface `ISLE_Reporter`.

3.2.6.4.1.1 The access violation alarm record shall be generated when a peer identifier is not registered in the configuration database of the proxy (see 3.2.6.2.4.2) or differs from the expected one. It shall comprise as much information as possible to allow investigation of the event. The information entered shall include but not be limited to the following:

a) For access violation associated with a BIND invocation, the alarm record shall contain the following parameters extracted from the PDU:

   1) the initiator identifier; and

   2) the service instance identifier.

b) For access violation associated with a BIND return, the alarm record shall contain the responder identifier extracted from the PDU and the following parameters derived from the attributes of the association:

   1) the responder port identifier; and

   2) the service instance identifier.

3.2.6.4.1.2 The authentication alarm record shall be generated whenever authentication fails. It shall comprise as much information as possible to allow investigation of the event. The information entered shall include but not be limited to the following:

a) the peer identifier;

b) the credentials for which authentication has failed; and

c) the service instance identifier.

3.2.7 PASS-THROUGH MODE OF OPERATION

3.2.7.1 As an optional feature, the proxy shall support a special 'pass-through' mode of operation, in which processing of the proxy is modified as defined by the following specifications. The means by which this mode of operation shall be enabled is defined and documented by the implementation.

NOTES

1 The pass-through mode is required for a gateway in order to support end-to-end identification and authentication, and preservation of other parameters set by the proxy in end-systems.

2 An implementation might support enabling and disabling of the pass-through mode by a parameter in the configuration database or might provide a special version of the proxy for use in a gateway.

3.2.7.1.1 The proxy shall not insert the local application identifier into a BIND invocation or a BIND return received from its local client, but use the parameter in the operation object. If the proxy needs to generate a BIND return on its own behalf (see 3.2.6.2.1.2), it shall insert the local application identifier as responder identifier. In these cases, the BIND return shall not include credentials.

NOTE – This modifies 3.2.6.2.2 and 3.2.6.2.3.4.

3.2.7.1.2 When receiving a BIND invocation from its local client, the proxy shall use the initiator identifier in the operation object, to determine the authentication mode. The peer identifier for the association shall remain undefined initially.

NOTE – This modifies 3.2.6.2.3. The peer identifier is assigned to the association as specified in 3.2.7.1.3. The procedure specified here implies that all responders linked to one initiator must have the same authentication mode. In cases where this constraint is not acceptable, different initiator identifiers must be used for different authentication modes.

3.2.7.1.3 The proxy shall process a BIND return PDU received from the peer proxy on an association in the state 'bind pending' as defined by the following specifications.

NOTE – This modifies 3.2.6.2.4.

3.2.7.1.3.1 If the responder is not registered, it shall abort the association with the diagnostic 'access denied' and generate an access violation alarm.

3.2.7.1.3.2 If the responder is registered, but the authentication mode differs from the one assigned to the association, the proxy shall abort the association with the diagnostic 'other reason'.

3.2.7.1.3.3 If the responder is registered and the authentication mode matches the one assigned to the association, the proxy shall assign the identifier and the security attributes of the responder to the association.

3.2.7.1.4 The proxy shall not generate credentials for PDUs transmitted to the peer proxy but use the credentials parameter of the operation object passed by its client.

NOTE – This modifies 3.2.6.3.3. Note that the proxy shall perform authentication of PDUs as defined in 3.2.6.3.1.

3.2.7.2 In the pass-through mode of operation, dynamic port registration shall not be required. The lack of dynamic port registration may imply restrictions, which must be clearly documented for an implementation.

NOTES

1 This modifies 3.2.5.1.

2 On a gateway, the responder port identifier is not known in advance of an incoming BIND invocation. An example for restrictions that might be implied is that only a single instance of the proxy can be used within the gateway and all associations bound via the same port have to be handled by a single process.

3.2.8 LOGGING AND NOTIFICATION

3.2.8.1 The proxy shall generate log messages for important events and enter them to the system log of the hosting system using the interface `ISLE_Reporter`, passed to its configuration method.

NOTE – The arguments to be supplied with a log message are specified in A9.2. Specific requirements and constraints on how this interface must be used are defined in 3.6.2.

3.2.8.2 The log messages generated by the proxy shall include, but not be limited to those explicitly defined in this section.

NOTE – Guidelines for selection of the events that are entered to the log are defined in 3.6.2.

3.2.8.3 The proxy shall notify the application of the events defined in 3.6.2.6.

3.2.8.4 The log messages and notifications shall be defined and documented by implementations of the API Proxy.

3.2.8.4.1 Each log message shall be identified by a unique number, which is referenced in the documentation and passed to the interface `ISLE_Reporter`, when the message is logged.

3.2.8.4.2 Log message identifiers in the range 0 to 999 shall be reserved for use by this Recommended Practice and supplemental Recommended Practice documents for service-specific APIs. These log message identifiers shall not be used for messages defined by implementations.

NOTE – This version of the specification does not define any log messages. Identifiers are reserved for potential future use. Beside this constraint, each component implementation can independently assign log message identifiers, as the component identification is also passed to the interface `ISLE_Reporter`.

3.2.9 DIAGNOSTIC TRACES

NOTE – Support for diagnostic traces is an optional feature. This subsection contains specific requirements for the proxy. Further general specifications concerning the events that are traced and the information entered in trace records are provided in 3.6.3.

3.2.9.1 The proxy shall provide a feature to generate trace records for events and to pass them to the interface `ISLE_Trace`.

3.2.9.1.1 A trace for a specific association can be started and stopped via the interface `ISLE_TraceControl` provided by association objects.

3.2.9.1.2 Traces started via the interface `ISLE_TraceControl` exported by the proxy shall include all associations as well as events that cannot be associated with a specific association. When tracing is stopped via the interface of the proxy, all traces started for individual associations shall be stopped as well.

3.2.9.1.3 After creation, all traces in the proxy shall be disabled.

3.2.10 EXECUTION CONTEXT

3.2.10.1 Processes

3.2.10.1.1 Every instance of the proxy component shall exist within a single application process and provide interfaces only to other components in the same process.

3.2.10.1.2 For a given proxy implementation, the creator function shall ensure that a single instance of the proxy component is created within one process. Proxy implementations shall be distinguished by the protocol they use for communication with the peer proxy. The protocol identifier shall be available via the administrative interface of the proxy.

NOTE – Several proxies using different communications technologies or different protocols can exist within one process. The protocol identifier is required to distinguish different proxies and to route outgoing BIND invocations to the correct proxy instance.

3.2.10.1.3 The proxy is able to support the following configuration of processes, service instances, and communication ports, for associations in the initiator role and for associations in the responder role.

NOTE – The meaning of the term communication port depends on the technology. It can refer to an end-point of a transport or session connection, an object reference, or any other address or routing information by which PDUs are routed to a given process.

3.2.10.1.3.1 A process shall handle all service instances using one or more communication ports.

3.2.10.1.3.2 Service instances using the same communication port may be distributed to different existing processes in a manner defined by the application.

NOTES

1 Depending on the technology and its specific use by the proxy, the configuration defined in 3.2.10.1.3.2 can require a separate process handling all or part of the interface to the communications service provider. Such processes are considered part of the proxy infrastructure. Routing of BIND invocations might be based on technology-specific addressing or on the service instance identifier. The dynamic

port-registration procedure defined in 3.2.5 can be used to establish the associated routing path.

2 Reference [J3] provides examples for architectures that can be supported by a proxy implementing these specifications.

3 This version of the specification does not include launching of application processes as a result of reception of a BIND invocation. Use of this technique is not excluded, but cannot be expected from a conforming implementation.

3.2.10.2 In Process Threads

3.2.10.2.1 For the interaction with its client, the proxy shall provide at least one of the behaviors defined in 3.7 as well as the control interface associated with the provided behavior.

3.2.10.2.1.1 An implementation may provide more than one of the specified behaviors. If that option is selected the implementation shall specify the means by which the desired behavior can be selected.

NOTE – An implementation may support selection of the behavior by off-line configuration, or may support dynamic selection by call of a specific start function.

3.2.10.2.1.2 The proxy shall provide the same behavior for all exported interfaces.

3.2.10.2.1.3 The proxy shall expect that the complementary interfaces provided by the client and used by the proxy have the same behavior.

3.2.11 CONFIGURATION

3.2.11.1 The proxy can be configured for a specific deployment environment by definition of parameters in a configuration database.

NOTE – For the operation of the proxy, an implementation may additionally require that supporting programs or external systems have been installed, configured in a specific manner, and have been started. Such requirements must be documented for an implementation together with a reference to the relevant installation and operating instructions.

3.2.11.1.1 The detailed content, the structure, and the format of the configuration database are implementation specific and are documented for an implementation together with the procedures for entry and update of configuration parameters.

NOTE – This specification does not prescribe how the database is created and how it is accessed. The configuration database may consist of a simple text file or a set of files, or it may be distributed to one or more directory systems or management information databases.

3.2.11.1.2 Modification of the configuration database shall be considered a maintenance activity. The procedures for entry and update of configuration parameters may require the proxy to be terminated and restarted for the modifications to become effective.

3.2.11.1.3 The proxy shall specify a configuration file, which provides all information required by the proxy to access the configuration database. The full path name of this file shall be passed to the `Configure()` method of the administrative interface.

NOTE – The configuration file might contain all configuration parameters or might contain references to other files or information that enables the proxy to query some database for the configuration parameters.

3.2.11.2 The information in the configuration database of the proxy shall include but not be limited to:

a) configuration parameters required for configuration and initialization of the data communications system such as local addresses, operational modes, etc.;

b) the acceptable delay between failure of the data communication service and the associated report by the proxy;

c) a specification whether the proxy shall support associations in the responder role, in the initiator role, or both;

d) mapping of the logical responder port identifiers specified by CCSDS to address information;

e) one or more local responder port identifiers on which a proxy supporting associations in the responder role shall accept BIND invocations;

f) the list of SLE service types supported by all components in the installation, and for each service type the list of version numbers that can be supported;

NOTE – Version negation is not supported by this Recommended Practice, see 1.2.2 item a).

g) the identifier and security attributes of the local SLE application;

h) the list of registered peer applications including, for each application, the identifier, authentication mode, and the security attributes (if used);

i) the acceptable delay between the time credentials have been created and the time of authentication;

j) the maximum number of incoming PDUs that shall be queued according to the definitions in 3.2.3.1;

k) the maximum number of PDUs that shall be queued for transmission;

l) the mode of operation, with the possible values ‘default’ and ‘pass-through’.

NOTE – The pass-through mode of operation is defined in 3.2.7.

3.2.12 INITIALIZATION, START-UP, TERMINATION, AND SHUTDOWN

3.2.12.1 The proxy must be initialized by a call to the method `Configure()` of the interface `ISLE_ProxyAdmin`, passing it the name of the configuration file and the interfaces of other components.

NOTE – The interfaces needed by the proxy and the exact signature are defined in A7.

3.2.12.1.1 When the method `Configure()` is called, the proxy shall check the configuration on completeness and consistency. If any of the checks fail, the proxy shall generate appropriate error messages to the log and return a result code indicating a configuration error.

3.2.12.1.2 If the configuration database is correct, the proxy shall perform all actions required to configure itself and then any data communication products it uses. If there are any errors, the proxy shall log errors indicating the reason and return an error code to the client.

3.2.12.1.3 Only when all initialization procedures have been completed successfully and the proxy is ready for operation, it shall return a positive result code.

3.2.12.2 Operation of the proxy shall be started by a call to the start method associated with the behavior selected according to 3.2.10.2.1. The proxy shall only start the operation if the initialization has been completed with success. Otherwise, the method shall return an error.

NOTES

1 The start method is defined by the control interface associated with the selected behavior. Control interfaces are specified in A6.1.1 and A6.1.6.

2 The specification implies that a proxy supporting associations in the responder role shall start listening for incoming BIND invocations only after call of the start method.

3.2.12.3 Operation of the proxy shall be terminated by a call to the terminate method associated with the behavior selected according to 3.2.10.2.1. When this method is called, the proxy shall stop processing and revert to the state it had after configuration and before the start method was invoked.

NOTE – The terminate method is defined by the control interface associated with the selected behavior. Control interfaces are specified in A6.1.1 and A6.1.6.

3.2.12.3.1 If any associations are still active, the proxy shall abort these associations.

3.2.12.3.2 The proxy shall stop listening on the network interface, if applicable, and release all resources it has allocated after the call to the start function.

3.2.12.3.3 A proxy must expect that other proxies using the same communication infrastructure exist on the system and must make sure that their operation is not affected by termination activities.

3.2.12.4 The proxy shall be instructed to shutdown by a call to the method `ShutDown()` in its administrative interface.

3.2.12.4.1 The proxy shall reject the request when it is still operating.

NOTE – In this case, the client shall be required to invoke the terminate method first.

3.2.12.4.2 The proxy shall release all interfaces of other components, to which it still holds references, delete all internal objects, and release any other resources it has allocated.

3.2.12.4.3 The proxy shall ensure that all objects of the component are deleted when clients holding a reference to interfaces have released these references.

3.2.12.4.4 When the method returns with success, the proxy has ceased to exist.

3.2.13 COMPONENT OBJECTS AND INTERFACES

3.2.13.1 The component API Proxy shall implement the following component objects and interfaces:

NOTE – Component objects are defined in annex D of this specification. As explained there, a component object is an externally visible entity that may be implemented by a single object or by several internal objects, which co-operate to provide the required external view. As specified in annex D, every component object supports the interface `IUnknown` in addition to the interfaces listed in this subsection. The interfaces referenced in this subsection are specified in annex A.

a) a single instance of the API Proxy, which shall export the following interfaces and support navigation between these interfaces via the method `QueryInterface()`:

   1) the interface `ISLE_ProxyAdmin`;

   2) the interface `ISLE_AssocFactory` if the proxy supports associations in the initiator role;

   3) the interface `ISLE_Sequential` if the proxy supports ‘sequential interface behavior’ as specified in 3.7.2;

   4) the interface `ISLE_Concurrent` if the proxy supports ‘concurrent interface behavior’ as specified in 3.7.2; and

   5) the interface `ISLE_TraceControl` if the proxy supports diagnostic traces as specified in 3.2.9;

b) association objects, which shall export the following interfaces and support navigation between these interfaces via the method `QueryInterface()`:

NOTE – A separate object shall be provided for every data communication association.

1) the interface `ISLE_SrvProxyInitiate`; and

2) the interface `ISLE_TraceControl` if the proxy supports diagnostic traces as specified in 3.2.9;

c) one or more objects for processing of external events, which shall export the interface ISLE_EventProcessor if the proxy supports ‘sequential interface behavior’;

d) one or more objects for processing of a timeout, which shall export the interface ISLE_TimeoutProcessor if the proxy supports ‘sequential interface behavior’.

3.3 API SERVICE ELEMENT

3.3.1 FEATURES

3.3.1.1 The service element shall create, configure, maintain, and delete service instances, see 3.3.2.

3.3.1.2 The service element shall provide interfaces to generate operation objects with initialized parameters according to the configuration of a service instance, see 3.3.3.

3.3.1.3 The service element shall handle binding and unbinding of service instances, see 3.3.4.

3.3.1.4 The service element shall enforce conformance of the protocol data units exchanged to the state tables defined in the CCSDS Recommended Standards for SLE transfer services as far as these do not refer to events and procedures related to service production.

NOTE – A detailed specification of the state tables processed by the service element is provided in section 4. These state tables complement the specifications in this section. They are considered mandatory for a conforming implementation.

3.3.1.5 The service element shall ensure that the parameters of SLE protocol data units conform to the specification in the CCSDS Recommended Standards for SLE transfer services.

NOTE – Processing of SLE Protocol Data Units is detailed in 3.3.5.1.

3.3.1.6 The service element shall handle invocation identifiers and timeout monitoring for operation returns, see 3.3.5.2.

3.3.1.7 The service element shall implement the transfer buffer for return link services including the procedures for discarding of buffered data in the delivery mode timely online and flow control for the delivery modes complete online and offline, see 3.3.5.3.

3.3.1.8 The service element shall implement flow control for TRANSFER-DATA invocations for forward link services, see 3.3.5.4.

3.3.1.9 For an SLE service provider, the service element shall provide an interface for updating service parameters and status parameters by the client, see 3.3.5.5.

3.3.1.10 The service element shall process GET-PARAMETER and SCHEDULE-STATUS-REPORT invocations received from the peer, see 3.3.5.5.

3.3.1.11 The service element shall generate entries to the log of the hosting system for all important events, see 3.3.6.

3.3.1.12 The service element shall provide a feature to produce an event trace for the complete service element and for individual service instances, see 3.3.7.

3.3.1.13 The service element shall support a range of execution environments with respect to use of in-process threads, see 3.3.8.

3.3.1.14 The service element shall use a configuration database, which controls its operation within a specific deployment environment, see 3.3.9.

3.3.2 SERVICE INSTANCE MANAGEMENT

3.3.2.1 Creation of Service Instances

3.3.2.1.1 A service instance shall be created on request of the application via the interface `ISLE_SIFactory`.

3.3.2.1.1.1 The request to create a service instance shall identify the service type and the role (service user or service provider). If the service element does not support the service type or the role, it shall reject the request.

NOTE – An implementation may support service instances in the user role and in the provider role concurrently. This feature enables an application to act as an SLE service user for one service instance and as an SLE service provider for another instance. However, an implementation may also restrict the role of the service instances supported to either ‘user’ or ‘provider’.

3.3.2.1.1.2 For services in the user role, the request to create a service instance additionally shall identify the version number of the specified service type. If the service element does not support the specified version, it shall reject the request.

NOTES

1 The service instance shall use the version number to generate the BIND invocation with the desired version number. Service instances in the provider role shall obtain

the version number from the incoming BIND invocation. Checking of the version number on the provider side is specified in 3.2.4.2.2.

2 The service types and versions supported by an implementation shall be identified in the implementation specific documentation.

3.3.2.2 Configuration of Service Instances

3.3.2.2.1 When a service instance has been created, it must be configured using the interface `ISLE_SIAdmin`.

3.3.2.2.1.1 For service instances in the responder role, additional configuration parameters must be supplied via the service-type specific administrative interface specified by the relevant supplemental Recommended Practice for the service-specific API.

3.3.2.2.1.2 The configuration parameters that must be set via the interface `ISLE_SIAdmin` are:

a) the service instance identifier;

b) the peer identifier;

NOTE – The peer identifier is either the initiator identifier or the responder identifier. If the service instance acts as an initiator in the BIND operation, the peer identifier is used to check the parameter ‘responder identifier’ in the BIND return PDU. If the service instance acts as a responder, the peer identifier is used to check the parameter ‘initiator identifier’ in the BIND invocation PDU.

c) the scheduled provision period defined by the start time and the stop time;

d) the initiator of the BIND operation (service user or service provider);

e) the responder port identifier; and

f) the value of the timeout in which returns must arrive for confirmed operations.

NOTE – These parameters are defined in the CCSDS Recommended Standards for SLE transfer services.

3.3.2.2.1.3 Configuration of a service instance shall be terminated by a call to the method `ConfigCompleted()` of the interface `ISLE_SIAdmin`. This method shall check the configuration parameters on completeness and consistency and reject the configuration if a deficiency is detected.

NOTE – Configuration parameters must not be modified after a successful return of the method `ConfigCompleted()`. The effect of an attempt to set a parameter when the initial configuration has completed is undefined.

3.3.2.2.1.4 The checks performed for the common configuration parameters passed via the interface `ISLE_SIAdmin` shall include the following:

a) the service instance identifier must be valid and unique for all service instances currently handled by the service element;

NOTE – The validity of a service instance identifier is verified by the component 'SLE Utilities' via the interface `ISLE_SII`.

b) for a service instance in the provider role the start and end time for the scheduled provision period must be specified or must be set to NULL;

c) if the start time is set to NULL, the service instance shall assume that the provision period begins after invocation of the method ConfigCompleted();

d) if the end time is set to NULL, the service instance shall assume that the provision period never expires;

e) if the start and end time are specified, the start time must be earlier than the end time;

f) if the end time is specified the end time must not be in the past;

NOTE – A service instance in the user role shall not constrain the application with respect to the time the service is requested. It shall accept the scheduled provision period parameter if it is supplied, but shall not require it. If it is supplied, it shall not further process it.

g) the responder port identifier must be defined in the configuration database of the service element, if the service instance initiates the BIND operation.

NOTE – The responder port identifier is used by the service instance to select the proxy instance to which the BIND invocation shall be sent.

3.3.2.2.1.5 For service instances in the provider role, checks performed on the configuration shall include those defined for the specific service type in the relevant supplemental Recommended Practice for the service-specific API.

3.3.2.2.1.6 If the service element does not support an option related to one of the configuration parameters, the configuration shall be considered incorrect and shall be rejected.

NOTE – For instance, an implementation might not support provider-initiated binding.

3.3.2.2.1.7 If the service instance responds to BIND invocations, the service element shall register the responder port as part of the method `ConfigCompleted()` and reject the configuration if port registration is not accepted by the proxy.

NOTE – Port registration is defined in 3.2.5.

3.3.2.3 Deletion of Service Instances

3.3.2.3.1 A service instance shall only be deleted on request of the application.

3.3.2.3.1.1 If the state of the service instance is not 'unbound' at the time deletion is requested, the service element shall reject the request.

NOTE – The application will have to abort the association before deleting the service instance. The service element does not require that the scheduled provision period has terminated when a service instance is deleted. (The states of a service instance are defined in section 4.)

3.3.2.3.1.2 If the responder port has been registered at the proxy, the service element shall request de-registration of the port before the service instance is actually deleted.

3.3.2.3.1.3 When deleting the service instance, the service element shall release all resources allocated to the service instance as well as all interfaces used by the service instance.

3.3.2.4 End of Provision Period

When the scheduled provision period of a service instance supporting the provider role expires, the service element shall inform the application via the interface `ISLE_ServiceInform`.

NOTES

1 If the end time of the provision period was set to NULL during configuration of the service instance, the service element shall assume that the provision period never expires.

2 The service element shall not monitor the provision period for service instances in the user role and shall not inform the application when the period ends.

3.3.3 CREATION AND CONFIGURATION OF OPERATION OBJECTS

3.3.3.1 Service instances shall export the interface `ISLE_SIOpFactory` to create pre-configured operation objects.

3.3.3.1.1 A service instance shall only return operation objects which are defined for the service type and version it supports, and which are invoked by applications in the role it implements.

NOTE – This specification implies that a RAF service instance in the user role generates a START operation but does not generate a TRANSFER-DATA operation.

3.3.3.1.2 The service instance shall use the interface `ISLE_OperationFactory` exported by the component 'SLE Operations' to create the operation object, and set selected parameters of the operation object according to its own configuration.

NOTE – The parameters that are set by the service instance are defined for each operation object in annex A or in the supplemental Recommended Practice documents for service-specific APIs.

3.3.4 BINDING AND UNBINDING

3.3.4.1 User Initiated Binding

NOTE – Further details of the BIND and UNBIND operations are specified in the state tables for service instances in section 4. These state tables complement the following specifications. They are considered mandatory for a conforming implementation.

3.3.4.1.1 Service Instances in the User Role

3.3.4.1.1.1 A service instance supporting the user role and the BIND-initiator role shall initiate the BIND operation when receiving a BIND invocation request from the application in the state 'unbound' via the interface `ISLE_ServiceInitiate`. It shall initiate the UNBIND operation when receiving an UNBIND invocation in the state 'bound'.

3.3.4.1.1.2 The service instance shall create an association via the proxy interface `ISLE_AssocFactory` and use the interface `ISLE_SrvProxyInitiate` provided by the association for binding, unbinding and service provisioning.

NOTE – This specification does not prescribe when the association object is created. An implementation might create a new association when the BIND operation is initiated. Alternatively an implementation might create the association object when the service instance is created and use it throughout the lifetime of the service instance.

3.3.4.1.1.3 The configuration database of the service element shall contain a table mapping port identifiers to protocol identifiers. The service element shall use this table and the responder port identifier in the service instance to select the proxy instance, from which it will request creation of the association.

NOTE – The protocol identifier supported by a proxy is specified when the proxy is registered with the service element. The associated procedures are defined in 3.3.10.

3.3.4.1.1.4 The service element shall insert the peer identifier into the parameter 'responder identifier' of the BIND operation object passed to the proxy.

NOTE – This parameter is used by the proxy to retrieve information about the responder from its configuration database. In addition, the proxy shall check the identifier against the responder identifier in the BIND return PDU and take appropriate actions if these do not match. See 3.2.6.2 for further details.

3.3.4.1.1.5 The BIND and UNBIND operations shall be performed according to the general rules specified in 3.3.5.

3.3.4.1.2 Service Instances in the Provider Role

3.3.4.1.2.1 A service element supporting service instances in the responder role shall implement the interface `ISLE_Locator`, which shall be used by the proxy to notify the service element of a BIND invocation received via the network.

3.3.4.1.2.2 When receiving a notification of a BIND invocation via the method `LocateInstance()` of the interface `ISLE_Locator`, the service element shall analyze the PDU. If the BIND invocation is acceptable, it shall link the requested service instance with the association using the pointer to the interface `ISLE_SrvProxyInitiate`, and return a reference to the interface `ISLE_SrvProxyInform` of the service instance.

3.3.4.1.2.3 The service element shall perform the checks in 3.3.4.1.2.4 to 3.3.4.1.2.10 on the BIND invocation in the sequence specified. If any of the checks fail, it shall not pass the BIND invocation to the application, but reject the BIND invocation. If the checks are performed within the method `LocateInstance()`, this method shall return an appropriate error code. If the checks are performed by the service instance after having received the BIND invocation via the interface `ISLE_SrvProxyInform`, it shall generate a BIND return with a negative result and the appropriate diagnostic.

NOTE – This specification does not prescribe whether these checks are performed by the method `LocateInstance()` of the interface `ISLE_Locator`, or by the service instance when receiving the BIND invocation via the interface `ISLE_SrvProxyInform`. The checks must be performed before the BIND invocation is passed to the application and the appropriate method to reject errors must be applied.

3.3.4.1.2.4 The service instance identifier in the BIND invocation must match the identifier of an existing service instance. If the check fails, the BIND invocation shall be rejected with the diagnostic ‘no such service instance’.

3.3.4.1.2.5 The initiator identifier in the BIND invocation must match the peer identifier defined for the service instance. If the check fails, the BIND invocation shall be rejected with the diagnostic ‘service instance not accessible to this initiator’. In addition, the service element shall enter an ‘access violation alarm’ in the system log and notify the application using the interface `ISLE_Reporter`.

3.3.4.1.2.6 The information included into the access violation alarm shall include, but not be limited to:

a) the initiator identifier in the BIND invocation; and

b) the service instance identifier of the service instance.

NOTE – Some of this information can be conveyed by standard arguments of the `LogRecord()` function in the interface `ISLE_Reporter`. This information should not be duplicated in the record itself.

3.3.4.1.2.7 The service type of the service instance must match the one indicated in the BIND invocation. If the check fails, the BIND invocation shall be rejected with the diagnostic ‘inconsistent service type’.

3.3.4.1.2.8 The version number in the BIND invocation must be supported by the service instance. If the check fails, the BIND invocation shall be rejected with the diagnostic ‘version not supported’. Otherwise the service instance shall memorize the version number and ensure that the service is provided as specified for that version.

NOTE – As the API Proxy already checks the version number against the versions identified in its configuration database (see 3.2.4.2.2.5), reception of an unsupported version number by the service instance is an indication of a configuration problem. Therefore, implementations should issue an alarm if that happens.

3.3.4.1.2.9 The time of the request must be within the scheduled provision period of the service instance. If the check fails, the BIND invocation shall be rejected with the diagnostic ‘invalid time’.

3.3.4.1.2.10 The state of the service instance must be UNBOUND. If the check fails, the BIND invocation shall be rejected with the diagnostic ‘already bound’.

3.3.4.1.2.11 The UNBIND operation for a service instance in the responder role shall be performed according to the general rules specified in 3.3.5. The additional specifications in 3.3.4.1.2.12 to 3.3.4.1.2.14 shall apply to a service instance supporting the provider role.

3.3.4.1.2.12 If the parameter ‘unbind reason’ in the UNBIND invocation is set to ‘suspend’, the following steps shall be performed. The state of the service instance shall be set to ‘unbound’, the service parameters shall be reset to the initial state, if applicable, and the service instance is ready to receive a new BIND invocation.

NOTE – Handling of the service parameters in the case of an UNBIND is specified individually for the every service type.

3.3.4.1.2.13 If the parameter ‘unbind reason’ in the UNBIND invocation is set to ‘end’, the service element shall inform the application that the scheduled provision period has been prematurely terminated.

NOTE – Further BIND invocations shall be rejected with the reason ‘invalid time’, if the service instance has not yet been deleted by the application.

3.3.4.1.2.14 When the scheduled provision period of a service instance supporting the provider role ends and the state of the service instance is not ‘unbound’, the service element shall abort the association.

3.3.4.2 Aborting Associations

3.3.4.2.1 The service element shall abort an association in the following cases:

a) the application invokes the PEER-ABORT operation;

b) abort of the association is explicitly required by any other specification for the service element in this document; or

c) the service element cannot continue processing of the association, because it is affected by major problems.

NOTE – This specification implies that the proxy might abort the association in the case of a catastrophic failure also when that case is not specified in this document.

3.3.4.2.2 Whenever the service element aborts an association, it shall also forward a PEER-ABORT invocation to the application. It shall set the parameter ‘originator’ to ‘service element’ in the operation objects passed to the proxy and to the application.

3.3.4.2.3 Whenever an association used by a service instance in the provider role is aborted, for any reason and by any party, the state of the service instance shall be set to ‘unbound’. The service parameters shall be reset to the initial state, if applicable, and the service instance is ready to receive a new BIND invocation.

3.3.4.3 Releasing Resources

3.3.4.3.1 Following completion of the UNBIND operation and following an abort of the association, the service element shall release the resources allocated to the association as defined by the following specifications.

3.3.4.3.1.1 For a service instance in the provider role, the service element shall release the interface of the association object provided by the proxy.

NOTE – A proxy supporting associations in the responder role creates a new association for every incoming bind request and deletes the association object when the association has terminated. To enable final deletion, the service element must release all interfaces.

3.3.4.3.1.2 For a service instance in the user role, the service element shall release the association interfaces and request the proxy to delete the association object when it is no longer needed.

NOTE – This specification does not prescribe when the association object in the initiator role is deleted. Implementations are free to use a single association object during the lifetime of the service instance, or a new association for every BIND. An implementation is required to release an association object that has been created for a service instance latest when the service instance itself is deleted.

3.3.4.3.1.3 The service element shall clear the list of pending local and remote returns, cancel any timers still running, and release all operation objects on which it holds references.

3.3.4.3.1.4 If the service instance holds a transfer buffer, it shall release the buffer and all data it may still contain.

3.3.5 SERVICE PROVISIONING

3.3.5.1 Processing of SLE Protocol Data Units

3.3.5.1.1 Protocol Data Units Received from the Application

3.3.5.1.1.1 The service element shall accept SLE protocol data units in the form of operation objects from the application via the interface `ISLE_ServiceInitiate`. It shall process them as defined in this subsection and forward them to the proxy via the interface `ISLE_SrvProxyInitiate` for transfer on the association linked to the service instance.

3.3.5.1.1.2 The service element shall perform the checks in 3.3.5.1.1.3 to 3.3.5.1.1.6 on the PDUs received from the application. If any of the checks fails, the service element shall not forward the PDU to the proxy but reject the PDU by an appropriate return code to the function.

3.3.5.1.1.3 The PDU must be defined for the service type supported by the service instance and must be compatible with the role of the service instance (user or provider).

3.3.5.1.1.4 The parameters passed with the operation object must be complete, in range, and consistent with the configuration of the service instance.

NOTES

1 An example for a consistency check is the verification that the service type in a BIND invocation matches the type of the service instance.

2 Operation objects are required to perform all checks that can be done on the data passed without any further knowledge about the context and to provide an interface by which these checks can be invoked. That feature should be used by the service

element. However, it must be considered that some checks require further knowledge about the state and configuration of the service instance. These checks cannot be performed by the operation objects. The checks performed by the operation objects are defined in annex A and in the supplemental Recommended Practice documents for service-specific APIs.

3 The checks do not include those parameters that are handled by the service instance itself. If such parameters are checked by the operation object, they must be correctly set before checking is requested.

3.3.5.1.1.5 A return for a confirmed operation must be conveyed by an operation object that has previously been passed to the application by the service instance.

3.3.5.1.1.6 The PDU must be valid in the state of the service instance.

NOTE – In a multithreaded environment, the application might not yet have become aware of a state change in some cases. If so, the return code does not indicate an error but informs the application of the state change. These cases are specifically marked in the state tables in section 4.

3.3.5.1.1.7 With a return code indicating success, the service element shall guarantee that the PDU has been accepted by the proxy for transmission.

NOTE – The following specifications (3.3.5.1.1.8 to 3.3.5.1.1.10) for processing of error cases also apply when a PDU has been generated by the service element itself. The statements referring to return codes do not apply in that case.

3.3.5.1.1.8 When the proxy rejects a PDU with a return code indicating that the transmission queue is full, the service element shall abort the association with the PEER-ABORT operation. In addition, it shall set the code returned to the application for the affected PDU to ‘overflow’.

NOTE – Because of the flow control mechanisms built into the API, queue overflow cannot be caused by transfer of space-link data units. It can only happen because of excessive generation of other events related to the production process or excessively high status reporting frequencies. In these cases, the application would have no other option for handling the problem.

3.3.5.1.1.9 When the proxy rejects the transfer request with a code that informs the service element of a state change, the service element shall return the corresponding code to the application and not abort the association.

NOTE – In such cases, it must be expected that the event causing the state change is already pending and will be available to the service element soon.

3.3.5.1.1.10 When the proxy rejects the transfer request with a code that indicates a protocol error, the service element shall abort the association. In addition, it shall generate a log

message providing as much information as possible to investigate the problem, and notify the application via the interface `ISLE_Reporter`.

3.3.5.1.1.11 When the service element has passed an unconfirmed invocation or a return to the proxy, it shall release the operation object holding the PDU.

3.3.5.1.1.12 With the exception of PEER-ABORT invocations and invocations that are inserted into the transfer buffer for return services, the service element shall ensure that PDUs received from the application are passed to the proxy in the sequence received.

NOTE – Handling of PEER-ABORT invocations is specified in 3.3.4.2. Buffering of PDUs for return services is specified in 3.3.5.3.

3.3.5.1.2 Protocol Data Units Received from the Proxy

3.3.5.1.2.1 The service element shall receive SLE protocol data units in the form of operation objects from the proxy and process them as specified in this subsection. If the invocations and returns are accepted by the service element, it shall forward them to the application unless specified differently for specific operations.

3.3.5.1.2.2 The service element shall perform the following checks on the PDUs received from the proxy. If any of the checks fails, the service element shall not forward the PDU to the application, but shall respond by rejecting the PDU locally (see 3.3.5.1.2.3), rejecting the PDU via the protocol (see 3.3.5.1.2.4), or by aborting the association.

NOTE – The type of response is defined for the individual tests.

a) The PDU must be defined for the service type supported by the service instance. If the check fails, the PDU shall be rejected locally.

b) The PDU must be compatible with the role of the service instance (user or provider). If the check fails, the service element shall abort the association with the diagnostic ‘protocol error’.

NOTE – The proxy is not aware of the user or provider role of the service instance and shall not check the PDUs for compatibility with this role.

c) The parameters passed with the PDU must be complete, in range, and consistent with the configuration of the service instance. If the check fails, the PDU shall be rejected via the protocol.

NOTE – See also the note on 3.3.5.1.1.4.

d) An invocation PDU must be consistent with the configuration of the service instance. If the check fails, the PDU shall be rejected via the protocol.

NOTE – An example for a consistency check for an invocation is the verification that a FSP service instance has invoke directive capability when receiving an INVOKE-DIRECTIVE invocation. Consistency checks are defined by the supplemental Recommended Practice documents for service-specific APIs.

e) A return PDU must be conveyed by an operation object that has previously been passed to the proxy by the service instance. If the check fails, the PDU shall be rejected locally.

f) The PDU must be valid in the state of service instance. If the check fails, the type of response shall depend on the PDU and on the state of the service instance.

NOTE – The type of response is defined in the state tables in section 4.

3.3.5.1.2.3 The service element shall reject a PDU locally by returning an error code to the function which passes the PDU. In this case, it shall not modify the state of the service instance.

3.3.5.1.2.4 The action taken by the service element to reject a PDU via the protocol shall depend on the PDU type.

NOTE – If the PDU is rejected via the protocol, the method passing the PDU shall return a result code indicating success.

a) For a confirmed invocation PDU, the service element shall generate a return with a negative result and the appropriate diagnostic and forward this to the proxy for transmission.

b) For an unconfirmed invocation PDU or a return PDU, the service element shall abort the association with PEER-ABORT and the appropriate diagnostic.

3.3.5.1.2.5 When the service element has passed an unconfirmed invocation or a return to the application, it shall release the operation object holding the PDU.

3.3.5.1.2.6 With the exception of PEER-ABORT invocations, the service element shall ensure that PDUs received from the proxy are passed to the application in the sequence received.

NOTE – Handling of PEER-ABORT invocations is specified in 3.3.4.2.

3.3.5.2 Processing of Confirmed Operations

3.3.5.2.1 The service element shall process invocations of confirmed operations issued by the application as follows:

a) The service element shall assign an invocation identifier according to the rules specified in the CCSDS Recommended Standards for SLE transfer services and pass it to the operation object.

NOTE – It is noted that the confirmed operations BIND and UNBIND do not carry invocation identifiers. Therefore, this specification does not apply to these operations.

b) When the operation object has been accepted by the proxy for transmission, the service element shall place the object on a list of pending remote returns. In addition, it shall start a return timer for the object.

NOTE – The timeout value shall be passed to the service instance as part of its configuration.

c) When the proxy returns the operation object via the interface `ISLE_SrvProxyInform`, the service element shall cancel the return timer, remove the object from the list of pending remote returns, forward it to the application, and release the object.

d) If the return timer expires, the service element shall abort the association with the diagnostic 'return timeout'.

3.3.5.2.2 The service element shall process invocations of confirmed operations issued by the proxy as follows:

a) The service element shall verify that the invocation identifier of the operation object is unique for all operation objects on the list of pending local returns. If this check fails, the service element shall add a negative result and a diagnostic 'duplicate invocation identifier' to the object and forward it to the proxy for transmission.

NOTE – It is noted that the confirmed operations BIND and UNBIND do not carry invocation identifiers. Therefore, this specification does not apply to these operations.

b) The service element shall add the operation to the list of pending local returns.

c) When the application returns the operation object via the interface ISLE_ServiceInitiate, the service element shall remove the object from the list of pending local returns, forward it to the proxy for transmission, and release the object.

3.3.5.3 Buffering and Flow Control for Return Link Services

3.3.5.3.1 General Specifications

3.3.5.3.1.1 Service instances for return link services shall implement the transfer buffer defined by the CCSDS Recommended Standards for SLE return link services (references [4], [5] and [6]).

NOTE – The specification of the procedure assumes use of a single transfer buffer. An implementation may use multiple buffers to increase performance. However, an implementation must ensure that only a single buffer is queued for transmission in the delivery mode timely online.

3.3.5.3.1.2 The service element shall use the operation object TRANSFER-BUFFER for buffering and for transfer of the buffer to and from the proxy.

3.3.5.3.1.3 The size of the transfer buffer and the value of the release timer are configuration parameters passed to the service instance during configuration. The associated interface is defined by the supplemental Recommended Practice documents for return link service-specific APIs.

NOTE – The size of the transfer buffer is defined by the number of PDUs that can be inserted into the buffer.

3.3.5.3.1.4 A service instance in the provider role shall buffer and transmit data as follows:

NOTE – Variations of this basic procedure depending on the delivery mode are specified in 3.3.5.3.2.1, 3.3.5.3.3.1, and 3.3.5.3.4.1.

a) The service element shall insert TRANSFER-DATA invocations and SYNC-NOTIFY invocations into the buffer in the sequence received from the application.

b) When the buffer is full, or when the SYNC-NOTIFY invocation indicates ‘end of data’, the service element shall forward the complete buffer to the proxy requesting it to issue a notification when the buffer has been transmitted.

NOTE – Notifications for transmitted PDUs are specified in 3.2.3.2.

c) When the proxy cannot transmit the buffer immediately, the service element shall memorize that a buffer is queued until it receives the notification via the method `PDUTransmitted()`.

d) After transmission of the transfer buffer, the service element shall create a new buffer.

3.3.5.3.2 Delivery Modes Timely Online and Complete Online

3.3.5.3.2.1 For the delivery modes timely online and complete online, a service instance in the provider role shall handle the release timer and the associated procedure defined in the CCSDS Recommended Standards for return link services (references [4], [5] and [6]).

a) The value of the release timer is a configuration parameter passed to the service instance via the service-type specific interface defined in the supplemental Recommended Practice documents for return link service-specific APIs.

b) When the service element inserts the first PDU into an empty transfer buffer, it shall start the release timer.

c) When the release timer expires and the transfer buffer is not empty the service element shall transmit the buffer regardless of its fill-grade.

3.3.5.3.3 Delivery Mode Timely Online

3.3.5.3.3.1 For the delivery mode timely online, a service instance in the provider role shall apply the following additional rules before transferring the buffer to the proxy:

a) When a buffer is already queued, the service element shall request the proxy to discard the buffer.

b) If the return code from the proxy indicates that a buffer has been discarded, the service element shall insert a SYNC-NOTIFY invocation, indicating 'data discarded due to excessive backlog', at the beginning of the buffer.

NOTE – If data transfer for the previous buffer has already started or when the notification that the buffer has been transmitted is already pending in a different thread of control, the proxy shall indicate that no buffer has been discarded.

3.3.5.3.4 Delivery Modes Complete Online and Offline

3.3.5.3.4.1 For the delivery modes complete online and offline, a service instance in the provider role shall perform the following procedure to suspend data transfer when the transmission capacity is exceeded.

a) When the buffer is due for transfer and a buffer is already queued, the service element shall return a code to the application indicating 'suspend data transfer'.

b) When receiving PDUs that must be buffered from the application in a period, in which data transfer has been suspended, the service element shall reject the request with a code indicating 'data transfer suspended'.

c) When data transfer has been suspended and the service element receives the notification that the previous buffer has been transmitted, it shall forward the current buffer to the proxy, create a new buffer, and notify the application that data transfer can be resumed.

NOTE – The notification to the application is passed by the method `ResumeDataTransfer()` of the interface `ISLE_ServiceInform`.

3.3.5.3.5 STOP Operation

3.3.5.3.5.1 When receiving a STOP return from the application, a service instance in the provider role shall transmit the transfer buffer if it is not empty.

a) For the delivery mode timely online the service element shall apply the procedure defined in 3.3.5.3.3.1.

b) For the delivery modes complete online and offline, the service element shall transfer the buffer also when a buffer is already queued.

3.3.5.3.6 User Side Processing

3.3.5.3.6.1 When receiving a transfer buffer from the proxy, a service instance in the user role shall extract the TRANSFER-DATA invocations and SYNC-NOTIFY invocations and forward them to the application using individual operation objects. These objects shall be forwarded in the sequence the invocations have been stored into the buffer.

3.3.5.4 Flow Control for Forward Services

3.3.5.4.1 A service instance in the user role supporting a forward link service shall implement the following flow control procedure for the operation TRANSFER-DATA.

NOTE – The specification of the procedure assumes use of a single TRANSFER-DATA invocation pending for transmission by the proxy. An implementation may support multiple outstanding TRANSFER-DATA invocations to increase performance.

3.3.5.4.2 When receiving a TRANSFER-DATA invocation from the application, the service element shall forward it to the proxy requesting the proxy to issue a notification when the buffer has been transmitted.

NOTE – Notifications for transmitted PDUs are specified in 3.2.3.2.

3.3.5.4.3 When the proxy cannot transmit the buffer immediately, the service element shall return a code to the application indicating ‘suspend data transfer’.

3.3.5.4.4 When receiving a TRANSFER-DATA invocation in a period, in which data transfer has been suspended, the service element shall reject the request with a code indicating ‘data transmission suspended’.

3.3.5.4.5 When data transfer has been suspended and the service element receives the notification that the previous TRANSFER-DATA invocation has been transmitted, it shall notify the application that data transfer can be resumed.

NOTE – The notification to the application is passed by the method `ResumeDataTransfer()` of the interface `ISLE_ServiceInform`.

3.3.5.5 Handling of Service Parameters

3.3.5.5.1 A service instance in the provider role shall maintain all service parameters defined in the CCSDS Recommended Standards for SLE transfer services. For service parameters that are updated by the production process, the service instance shall provide an interface for the application to pass the updated values.

NOTE – The interface to update service parameters is service-type specific and is defined by the relevant supplemental Recommended Practice for the service-specific API.

3.3.5.5.2 A service instance in the provider role shall implement the GET-PARAMETER operation by returning the value of the requested parameter. It shall not forward the GET-PARAMETER invocation to the application.

3.3.5.5.3 A service instance in the provider role shall implement status reporting as defined in the CCSDS Recommended Standards for SLE transfer services.

3.3.5.5.3.1 The service element shall not forward the SCHEDULE-STATUS-REPORT invocation to the application, but perform all required actions internally.

3.3.5.5.3.2 When the SCHEDULE-STATUS-REPORT invocation requests periodic transfer of status reports, the service element shall check the cycle period against the limits specified in its configuration database and reject the request when the period is not within these limits.

3.3.5.5.3.3 When status reports have been scheduled, the service element shall generate the STATUS-REPORT invocation from the values of the service parameters at the time the status report is due and send it without involvement of the application.

3.3.5.5.3.4 If the delivery mode of the service instance is ‘offline’, the service element shall reject a SCHEDULE STATUS REPORT invocation with a negative return and the diagnostic ‘not supported in this delivery mode’.

NOTE – The CCSDS forward transfer services do not support the delivery mode ‘offline’. Therefore, ‘not supported in this delivery mode’ is not a valid diagnostic code for these services.

3.3.6 LOGGING AND NOTIFICATION

3.3.6.1 The service element shall generate log messages for important events and enter them to the system log of the hosting system using the interface `ISLE_Reporter`, passed to its configuration method.

NOTE – The arguments to be supplied with a log message are specified in annex A. Specific requirements and constraints on how this interface must be used are defined in 3.6.2.

3.3.6.2 The log messages generated by the service element shall include, but not be limited to those explicitly defined in this subsection.

NOTE – Guidelines for selection of the events that are entered to the log are defined in 3.6.2.

3.3.6.3 The service element shall notify the application of the events defined in 3.6.2.6.

3.3.6.4 The log messages and notifications shall be defined and documented by implementations of the API Service Element.

3.3.6.4.1 Each log message shall be identified by a unique number, which is referenced in the documentation and passed to the interface `ISLE_Reporter`, when the message is logged.

3.3.6.4.2 Log message identifiers in the range 0 to 999 shall be reserved for use by this Recommended Practice and its supplemental Recommended Practice documents for service-specific APIs. These log message identifiers shall not be used for messages defined by implementations.

NOTE – This version of the specification does not define any log messages. Identifiers are reserved for potential future use. Besides this constraint, each component implementation can independently assign log message identifiers, as the component identification is also passed to the interface `ISLE_Reporter`.

3.3.7 DIAGNOSTIC TRACES

NOTE – Support for diagnostic traces is an optional feature. This subsection contains specific requirements for the service element. Further general specifications concerning the events that are traced and the information entered in trace records are provided in 3.6.3.

3.3.7.1 The service element shall provide a feature to generate trace records for events and to pass them to the interface `ISLE_Trace`.

3.3.7.1.1 A trace for a specific service instance can be started and stopped via the interface `ISLE_TraceControl` exported by service instance objects.

3.3.7.1.2 When the argument `forward` in the method `StartTrace()` is set to true, the service instance shall start tracing for all associations to which it is bound as long as tracing is enabled. The service instance shall stop tracing by an association when `StopTrace()` is called, if it has started tracing by the association.

NOTE – The service instance can only forward the request if the proxy supports tracing.

3.3.7.1.3 Traces started via the interface `ISLE_TraceControl` exported by the service element shall include all service instances as well as events that cannot be associated with a specific service instance. When tracing is stopped via the interface of the service element, all traces started for individual service instances shall be stopped as well.

3.3.7.1.4 When the argument `forward` in the method `StartTrace()` is set to true, the service element shall start tracing for all proxies to which it is linked. The service element shall stop tracing by a proxy when `StopTrace()` is called, if it has started tracing by the proxy.

NOTE – The service instance can only forward the request if the proxy supports tracing.

3.3.7.1.5 After creation, all traces in the service element shall be disabled.

3.3.8 EXECUTION CONTEXT

3.3.8.1 Processes

3.3.8.1.1 Every instance of the component API Service Element shall exist within a single application process and provide interfaces only to clients in the same process.

3.3.8.1.2 The creator function shall ensure that a single instance of the service element component is created within one process.

3.3.8.2 In Process Threads

3.3.8.2.1 For the interaction with the application, the service element shall provide at least one of the behaviors defined in 3.7 as well as the control interface associated with the provided behavior.

3.3.8.2.1.1 The service element shall provide the same behavior for all interfaces exposed to the application.

3.3.8.2.1.2 The service element shall expect that the complementary interfaces provided by the application and used by the service element have the same behavior as the interfaces provided to the application.

3.3.8.2.2 For the interaction with the proxy, the service element shall provide at least one of the behaviors defined in 3.6 and control the proxy using the interface associated with that behavior.

NOTE – The behavior provided towards the proxy need not be the same as the one provided towards the application.

3.3.8.2.2.1 The service element shall provide the same behavior for all interfaces exposed to the proxy.

3.3.8.2.2.2 If the service element provides the sequential behavior towards the proxy, it shall provide the event monitor and the timer handler defined in 3.7.2.

NOTE – If the service element also provides sequential behavior towards the application, it must pass the event monitor and the timer handler supplied by the application to the proxy.

3.3.8.2.3 An implementation may provide more than one of the specified behaviors on either interface. If that option is selected, the implementation shall specify the means by which the desired behavior can be selected.

NOTE – An implementation may support selection of the behavior by off-line configuration, or may support dynamic selection by call of a specific start function.

3.3.9 CONFIGURATION

3.3.9.1 The service element can be configured for a specific deployment environment by definition of parameters in a configuration database.

3.3.9.1.1 The detailed content, the structure, and the format of the configuration database shall be implementation specific and documented for an implementation together with the procedures for entry and update of configuration parameters.

NOTE – This specification does not prescribe how the database is created or how it is accessed. The configuration database may consist of a simple file or a set of files, or it may be distributed to one or more directory systems or management information databases.

3.3.9.1.2 Modification of the configuration database shall be considered a maintenance activity. The procedures for entry and update of configuration parameters may require that the service element is terminated and restarted for the modifications to become effective.

3.3.9.1.3 The service element shall specify a configuration file, which provides all information required by the proxy to access the configuration database. The full path name of this file shall be passed to the `Configure()` method of the administrative interface.

NOTE – The configuration file might contain all configuration parameters or might contain references to other files or information that enables the service element to query some database for the configuration parameters.

3.3.9.2 The information in the configuration database of the service element shall include, but not be limited to:

a) the mapping of port identifiers to protocol identifiers for selection of the proxy instance, as defined in 3.3.4.1.1.3; and

b) the minimum and the maximum value for the reporting cycle to be supported for periodic status reports for a service element supporting the provider role.

3.3.10 INITIALIZATION, START-UP, TERMINATION, AND SHUTDOWN

3.3.10.1 The service element must be initialized by a call to the method `Configure()` of the interface `ISLE_SEAdmin`, passing it the name of the configuration file and the interfaces of other components needed.

NOTE – The interfaces needed by the service element and the exact signature are defined in annex A.

3.3.10.1.1 When the method `Configure()` is called, the service element shall check the configuration on completeness and consistency. If any of the checks fail, the service element shall generate appropriate error messages to the log and return a result code indicating a configuration error.

3.3.10.1.2 If the configuration database is complete and consistent, the service element shall perform all actions required to configure the component. If there are any errors, the service element shall log errors indicating the reason and return an error code to the client.

3.3.10.1.3 Only when all initialization procedures have been completed successfully it shall return a positive result code.

3.3.10.2 Following configuration, all proxy instances needed must be registered with the service element, using the method `AddProxy()` of the interface `ISLE_SEAdmin`.

3.3.10.2.1 The service element shall check whether the proxy registration is compatible with its configuration database, its capabilities, and previous registrations. If there are any problems, it shall log an error and reject the registration with an appropriate error code.

3.3.10.2.2 The checks performed by the service element shall include, but not be limited to the following:

a) The protocol identifier passed with the registration request must be defined in the configuration database if the proxy supports associations in the initiator role for the given deployment environment.

NOTE – The argument `role` of the method `AddProxy()` shall indicate the bind roles which associations of the proxy support for the given installation. If a proxy implementation can support the initiator role but this role is not needed by the application, this argument should be set to ‘provider only’.

b) The number of proxies registered must be within the limits supported by the service element.

c) Duplicate registration of the same proxy or the same protocol identifier must be prevented.

NOTE – Remaining checks must be performed when the service element is started. For instance, the service element must ensure that all protocol identifiers used in its mapping table are actually supported by a proxy that has been registered.

3.3.10.2.3 Registration of proxies must be performed after configuration and before operation of the service element is started. Invocation of the function at other times results in an error.

3.3.10.3 Operation of the service element shall be started by a call to the start method associated with the behavior selected according to 3.3.8.2.1.

NOTE – The start method is defined by the control interface associated with the selected behavior. Control interfaces are specified in annex A.

3.3.10.3.1 The service element shall only start operation when the initialization has been completed with success. Otherwise, the start function shall return an error.

3.3.10.3.2 As part of the start method, the service element shall start operation of all proxies that have been registered. If starting of any of the proxies fails, it shall log an error.

3.3.10.3.3 If starting of at least one of the proxies succeeds, the service element shall start operation. If starting of at least one of the proxies failed, the service element shall return a result indicating ‘degraded mode’. Otherwise, the function shall return with success.

NOTE – Start of a proxy may fail because of problems with one or more interfaces. In such a case, communication with a subset of the peer-systems might still be possible.

3.3.10.4 Operation of the service element shall be terminated by a call to the terminate method associated with the behavior selected according to 3.3.8.2.1. When this method is called, the service element shall stop processing and revert to the state it had after configuration and before the start method was called.

NOTE – The terminate method is defined by the control interface associated with the selected behavior. Control interfaces are specified in annex A.

3.3.10.4.1 If service instances are still active and not in the state ‘unbound’, the service element shall abort the associations and release the service instance objects.

3.3.10.4.2 The service element shall invoke the terminate method on all proxies which it has started.

3.3.10.4.3 The service element shall release all resources it has allocated after the call to the start method.

3.3.10.5 The service element shall be instructed to shutdown by a call to the method `ShutDown()` in its administrative interface.

3.3.10.5.1 The service element shall reject the request when it is still operating.

NOTE – In this case, the client shall be required to invoke the 'terminate function' first.

3.3.10.5.2 The service element shall release all interfaces of other components on which it still holds references, delete all internal objects, and release any other resources it may have allocated.

3.3.10.5.3 The service element shall ensure that all objects of the component are deleted when clients holding a reference on interfaces have released these references.

3.3.10.5.4 When the method returns with success, the service element has ceased to exist.

3.3.11 COMPONENT OBJECTS AND INTERFACES

3.3.11.1 The component API Service Element shall implement the following component objects and interfaces:

NOTE – Component objects are defined in annex D of this specification. As explained there, a component object is an externally visible entity that may be implemented by a single object or by several internal objects, which co-operate to provide the required external view. As specified in annex D, every component object shall support the interface `IUnknown` in addition to the interfaces listed in this subsection. The interfaces referenced in the following are specified in annex A and in the supplemental Recommended Practice documents for service-specific APIs.

a) a single instance of the API Service Element, which shall export the following interfaces and support navigation between these interfaces via the method `QueryInterface()`:

   1) the interface `ISLE_SEAdmin`;

   2) the interface `ISLE_Locator`;

   3) the interface `ISLE_SIFactory`;

   4) the interface `ISLE_Sequential` if the service element supports 'sequential interface behavior' as specified in 3.7.2;

   5) the interface `ISLE_Concurrent` if the service element supports 'concurrent interface behavior' as specified in 3.7.2; and

   6) the interface `ISLE_TraceControl` if the service element supports diagnostic traces as specified in 3.3.7;

b) service instance objects, which export the following interfaces and support navigation between these interfaces via the method `QueryInterface()`:

NOTE – A separate object shall be provided for every service instance created by the application.

1) the interface `ISLE_SIAdmin`;

2) the interface `I<SRV>_SIAdmin` for service instances in the provider role, if this interface is specified for the SLE service type supported by the service instance;

NOTE – The prefix `I<SRV>` is substituted by the abbreviation for the service type, e.g., '`IRAF`'. The supplemental Recommended Practice for the service-specific API defines the interface, if it is needed.

3) the interface `I<SRV>_SIUpdate` for service instances in the provider role, if this interface is specified for the SLE service type supported by the service instance;

NOTE – The prefix `I<SRV>` is substituted by the abbreviation for the service type, e.g., '`IFSP`'. The supplemental Recommended Practice for the service-specific API defines the interface, if it is needed.

4) the interface ISLE_SIOpFactory;

5) the interface ISLE_TraceControl, if the service element supports diagnostic traces as specified in 3.3.7; and

6) the interface ISLE_ServiceInitiate;

c) one or more objects for processing of external events, which shall export the interface `ISLE_EventProcessor` if the service element supports 'sequential interface behavior';

d) one or more objects for processing of a timeout, which shall export the interface `ISLE_TimeoutProcessor` if the service element supports 'sequential interface behavior'.

NOTE – If the service element supports the interface behavior 'sequential' on the interface to the proxy and the interface behavior 'concurrent' on the interface to the application, it shall also implement and export the interfaces `ISLE_EventMonitor` and `ISLE_TimerHandler.`

3.3.11.2 For the interface `ISLE_SrvProxyInform`, the service element shall implement one of the following options:

a) the interface `ISLE_SrvProxyInform` shall be exported by the same object exporting the interfaces listed in 3.3.11.1 item b); or

b) the interface `ISLE_SrvProxyInform` shall be implemented by a separate component object, which shall also implement a separate interface `IUnknown`.

NOTE – Clients of the component do not need to navigate between the interfaces listed in 3.3.11.1 item b) and the interface `ISLE_SrvProxyInform`. Therefore, an implementation may opt to support the interfaces to the proxy and to the application by different objects.

3.4 SLE OPERATIONS

3.4.1 OPERATION OBJECTS

3.4.1.1 The component 'SLE Operations' shall implement one operation object class for every SLE operation defined for the service types it supports. Common operations shall be provided by all implementations.

NOTE – Common operations are defined in this subsection. Service-type specific operations are defined in the relevant supplemental Recommended Practice for the service-specific API.

3.4.1.2 Operation objects shall store the parameters defined for the SLE operation, and provide read and write access to these parameters. For confirmed operations, operation objects shall contain the invocation parameters and the return parameters.

3.4.1.3 A reference to the interface `ISLE_Reporter` may be optionally passed as an argument to the creator function. This interface can be used by implementations to report errors and inconsistencies detected in the attributes of operation objects.

3.4.1.4 The component shall provide an 'operation factory', which shall create operation objects in response to requests received via the interface `ISLE_OperationFactory`.

3.4.1.4.1 A reference to the operation factory shall be returned by the creator function for the component.

3.4.1.4.2 The operation object to be created shall be specified by an identifier for the operation object interface, the operation type, the service type and the version number of the service. When the component does not support objects with the specified interface, the service type, or version number, or when the operation is not defined for the service type and version the factory shall reject the request.

3.4.1.4.3 Following creation, the parameters held by an operation object shall be set to the initial values defined in annex A for common operations and in the relevant supplemental Recommended Practice for the service-specific API for service-type specific operations.

NOTE – Deletion of operation objects shall be achieved by the reference counting scheme defined in annex D.

3.4.1.4.4 Unless specified differently in annex A, data passed to operation objects shall be considered the property of the operation object and be deleted when the object itself is deleted.

3.4.2 CHARACTERISTICS OF OPERATION OBJECTS

3.4.2.1 Common characteristics of operation objects shall be defined by the interface `ISLE_Operation`, which shall be inherited by all interfaces exported by operation objects.

NOTE – This subsection specifies essential characteristics but does not address every method of the interface. Annex A defines the methods.

3.4.2.1.1 An operation object shall be identified by the combination of an identifier for the operation, an identifier for the service type and the version number of the service type. Operation objects shall provide methods to query these identifiers.

3.4.2.1.2 Operation objects shall provide methods to verify that the invocation arguments are complete, consistent and in range.

NOTES

1 The checks are specified in annex A for common operations and in the relevant supplemental Recommended Practice for the service-specific API for service-type specific operations. These checks assume that the operation object has been passed from the proxy or is about to be passed to the proxy. Therefore, the checks do not include parameters handled by the proxy.

2 Implementations may issue log messages to report errors detected by these methods using the interface `ISLE_Reporter` passed to the creator function of the component.

3.4.2.1.3 All operation objects shall store the parameter for the invoker credentials.

3.4.2.1.4 Operation objects shall provide a method to produce a human readable string including the names and values of all parameters set in the object. For binary data, the method shall produce a dump of hexadecimal digits, where the maximum length of the dump is constrained by an argument to the method.

NOTE – This specification does not prescribe the format of the printout nor the names of the parameters. When defining the output it should be considered that it will be included into diagnostic traces and must be interpreted by humans. The output should be understandable to engineers that are not programmers.

3.4.2.1.5 In order to increase performance, the interfaces provided by operation objects are not safe in a multi-threaded environment. However, operation objects shall support an advisory lock that can be set on the object. This lock must be used by clients in a multi-threaded environment. The implementation shall prevent self-inflicting deadlocks.

3.4.3 CHARACTERISTICS OF CONFIRMED OPERATION OBJECTS

3.4.3.1 Common characteristics of objects implementing confirmed operations shall be defined by the interface `ISLE_ConfirmedOperation`, which is inherited by all interfaces exported by confirmed operation objects.

NOTE – This subsection specifies essential characteristics but does not address every method of the interface. Annex A defines the methods.

3.4.3.1.1 Confirmed operations shall store the invocation identifier used in the invocation PDU and in the return PDU.

3.4.3.1.2 Confirmed operations shall store the result of the operation and, if the result is set to 'negative', the associated diagnostics.

3.4.3.1.3 Confirmed operation objects shall store the parameter for the performer credentials.

3.4.3.1.4 Confirmed operation objects shall provide methods to verify that the return arguments are complete, consistent, and in range.

NOTE – The checks are specified in annex A for common operations and in the relevant supplemental Recommended Practice for the service-specific API for service-type specific operations. These checks assume that the operation object has been passed from the proxy or is about to be passed to the proxy. Therefore, the checks do not include parameters handled by the proxy.

3.4.4 COMMON OPERATION OBJECT CLASSES

3.4.4.1 Operations for Common Association Management

3.4.4.1.1 All implementations shall provide operation objects for the following SLE operations:

a) the operation BIND;

b) the operation UNBIND; and

c) the operation PEER-ABORT.

3.4.4.1.2 The operations for association management shall be included in the set of service-type specific operations provided for a given service type. The implementation shall ensure that the service type information returned by these objects matches the service type for which it has been created.

NOTE – This specification implies, for instance, that a BIND operation object that has been created for the FSP service always returns the service type identification 'FSP'.

3.4.4.2 Other Common Operations

3.4.4.2.1 All implementations shall provide objects for the following SLE operations, which are used for more than one SLE service type:

a) the STOP operation;

b) the SCHEDULE-STATUS-REPORT operation; and

c) the pseudo-operation TRANSFER-BUFFER.

NOTE – TRANSFER-BUFFER is actually not defined as an SLE operation by the CCSDS Recommended Standards for return link services (references [4] and [5] and [6]). Within the API, the operation object shall implement the transfer buffer used for return link services. It shall correspond to the PDU used for transmission of the transfer buffer.

3.4.4.2.2 These operations shall be included into the set of service-type specific operations provided for a given service type. The implementation shall ensure that the service type information returned by these objects matches the service type for which it has been created.

NOTE – This specification implies, for instance, that a STOP operation object that has been created for the FSP service always returns the service type identification ‘FSP’.

3.4.4.2.3 The operation object for the pseudo-operation TRANSFER-BUFFER shall provide a facility to queue and to de-queue any type of operation objects. The object shall not check what operation objects are inserted into the buffer.

NOTE – It is considered the responsibility of the client to insert only those objects for which buffering has been defined in the CCSDS Recommended Standards for return link services (references [4] and [5] and [6]). This implies that a client extracting objects from the buffer must verify that the type of the object is correct. Details on the features provided for queue handling are defined by the interface in annex A.

3.4.5 COMPONENT OBJECTS AND INTERFACES

3.4.5.1 The component SLE Operations shall implement the following component objects and interfaces:

NOTE – Component objects are defined in annex D of this specification. As explained there, a component object is an externally visible entity that may be implemented by a single object or by several internal objects, which co-operate to provide the required external view. As specified in annex D, every component object shall support the interface `IUnknown` in addition to the interfaces listed in this subsection. The interfaces referenced in the following are specified in annex A.

a) an object for the operation object factory exporting the interface `ISLE_OperationFactory`; and

b) operation objects for all operations required by the SLE service types supported.

NOTE – A separate object shall be supported for every operation object created via the operation factory.

3.4.5.2 Operation objects shall export the following interfaces and provide navigation between these interfaces via `QueryInterface()`:

a) the interface `ISLE_Operation`;

b) the interface `ISLE_ConfirmedOperation`, if the SLE operation supported by the object is confirmed; and

c) the interface specified for the operation type supported by the object.

3.5 SLE UTILITIES

3.5.1 GENERAL SPECIFICATIONS

3.5.1.1 The component 'SLE Utilities' shall implement auxiliary objects that must be passed across component boundaries. The object classes provided are:

a) a memory management class handling allocation and release of memory for data structures passed across component boundaries;

b) a time class handling specific CCSDS time formats;

c) a class handling the service instance identifier defined by the CCSDS Recommended Standards for SLE transfer services;

d) a class handling the credentials passed with SLE protocol data units for authentication of the peer identity; and

e) a class storing security attributes of an SLE application and capable of generating credentials and authenticating credentials.

NOTES

1 The API requires a common implementation for objects passed between component boundaries. Use of component interfaces for such objects instead of a standard class library minimizes the dependencies between the API components.

2 The functionality and the interfaces defined for SLE Utilities are minimal and restricted to what is needed for the SLE API.

3.5.1.2 The component shall provide a 'utility factory', which shall create utility objects in response to requests received via the interface `ISLE_UtilFactory`.

NOTE – Deletion of utility objects is achieved by the reference counting scheme defined in annex D.

3.5.1.3 A reference to the utility factory shall be returned by the creator function for the component.

3.5.1.4 An interface providing an external time source may be passed to the component as an argument to the creator function. If this option is used, the component shall use the external time source interface to obtain current time. Otherwise, it shall use system time.

NOTES

1 The external time source is provided via the interface `ISLE_TimeSource` defined in 3.6.4 and in annex A.

2 The current time obtained via the external time source or from system time is supplied to other API components via the Time class specified in 3.5.2.

3.5.1.5 In order to increase performance, the interfaces provided by utility objects are not safe in a multi-threaded environment, except for the memory management interface `IMalloc`.

NOTE – In general, utility objects shall either be stored locally or need to be accessed only in combination with the operation object passing the value. Therefore, access by a single thread of control can generally be guaranteed by the processing context. Should special protection be required, this must be implemented by the clients of the object.

3.5.2 MEMORY MANAGEMENT

NOTE – An SLE API specific memory management service is required to avoid inconsistencies between memory management services used by different independently developed components. The interface provided conforms to the COM memory manager specified in reference [J5] in order to allow use of the SLE API in a COM environment. However, implementations are not required to provide a COM conforming implementation and clients should only rely on the methods specified in this subsection. For further information see annex A and annex D.

3.5.2.1 The services of the memory manager shall be made available by the interface `IMalloc`. The features provided shall include:

a) allocation of a block of memory;

b) release of a previously allocated block of memory; and

c) re-allocation of a block of memory using a new block size, the contents of the block are unchanged up to the shorter of the new and old sizes.

3.5.2.2 Implementations may provide dummy implementations for the following methods defined by the interface `IMalloc`:

a) `GetSize()` always returning zero;

b) `DidAlloc()` always returning –1;

c) `HeapMinimize()`.

3.5.2.3 The component SLE Utilities shall make sure that all memory allocated and released via the interface `IMalloc` is subject to a consistent memory management scheme.

NOTE – Provided that this requirement is met, implementations may use multiple objects or a single object to implement the interface `IMalloc`.

3.5.2.4 The implementation of the interface `IMalloc` shall be multi-thread safe.

3.5.2.5 API components and SLE Applications using the API shall be required to use the memory manager for allocation and release of all data structures passed between API components and between API component and the SLE application.

NOTE – This requirement does not apply to utility objects and operation objects, as memory management for these is achieved by reference counting as specified in annex D.

3.5.3 TIME

NOTE – CCSDS SLE Recommended Standards require that all time parameters be in UTC. However, the time class is not required to perform conversion between local time and UTC. It is assumed that the systems providing or using SLE Services will use UTC as their system time or supply UTC time via the interface `ISLE_TimeSource`, if that interface is used. (For possible exceptions see 3.6.4.)

3.5.3.1 The services of the time class shall be made available by the interface `ISLE_Time`. The features provided shall include:

a) setting the time from the following inputs:

1) the CCSDS day segmented time code (CDS) specified in reference [1] with the following selection of options:

i) level 1 epoch, i.e., 01.01.1958;

ii) 16-bit day field;

iii) resolution in microseconds;

iv) the P-Field is implicit and not part of the input or output data;

2) the CCSDS ASCII Calendar Segmented Time Code specified in reference [1] with two variants A (Month/Day of Month) and B (Year/Day of Year), supporting the following subsets:

   i) the calendar subset;

   ii) the time subset to the resolution defined by the client;

b) output of the stored time in the formats defined by item a);

c) resetting the time to current time;

d) comparison of two time objects;

e) calculation of the difference between two time objects and output of the result in seconds and fractions of seconds.

3.5.3.2 After creation of an object, the time value shall be set to current time.

3.5.3.3 The time class shall support time values up to an accuracy of one microsecond.

NOTE – This requirement is to be understood such that the object shall maintain the accuracy of a time value passed to it. The accuracy of the time value when the current time is set depends on the capability of the platform.

3.5.4 SERVICE INSTANCE IDENTIFIER

NOTES

1 The service instance identifier is specified in the CCSDS Recommended Standards for SLE transfer services as a Distinguished Name as defined by reference [17]. In addition, the CCSDS Recommended Standards define a human readable string format. This class supports both formats and is able to convert between them.

2 This class is not required to provide a general implementation for distinguished names. In particular, the implementation may take advantage of the following:

- the attribute value is always an ASCII string;
- the values of the object identifiers used to identify the attributes have a fixed size and differ only in the last component.

3 [V1:] For version 1 of the SLE transfer services RAF, RCF, and CLTU, the specification of the Service Instance Identifier is provided in annex C of this Specification. By default the class shall support the service instance identifier format defined by the CCSDS Recommended Standards for transfer services. Support of the initial format defined by annex C is optional and must be requested by calling the appropriate method. Implementations not supporting the initial format shall return an error when this method is called.

3.5.4.1 The services of the service instance identifier class shall be made available by the interface `ISLE_SII`. The features provided shall include:

a) setting of the service instance identifier value from the following inputs:

1) a sequence of relative distinguished names where the attribute is identified by an object identifier presented as a sequence of integers and the attribute value as an ASCII string;

2) [V1:] for version 1 of the RAF, RCF or CLTU service, the standard ASCII representation of the service instance identifier as defined in annex C; or

3) [V2:] the standard ASCII representation of the service instance identifier as defined in the CCSDS Recommended Standards for SLE transfer services;

b) output of the service instance identifier in the formats defined in item a);

c) testing two service identifier objects for equality;

d) [V1:] for version 1 of the RAF, RCF or CLTU service, verification that the attributes used in the service instance identifier are those defined by annex C;

NOTE – For version 1 of the services RAF, RCF, and CLTU, the class shall not check the number, sequence, or selection of attributes. Also it shall not check any of the attribute values.

e) [V2:] verification that the format conforms to the specification provided in the CCSDS Recommended Standards for SLE transfer services.

NOTE – The attributes used in the identifier and the sequence of the attributes must conform to the specification and all required attributes must be present. In addition, the CCSDS Recommended Standards define a permissible set of values for some of the attributes, which must be adhered to.

3.5.4.1.1 The class shall process input and output as follows:

a) When processing input of a value presented in ASCII, the class shall check the syntax and perform the checks specified in 3.5.4.1. If there is an error, it shall reset the internal value to NULL and return an error.

b) In the ASCII representation produced as output, a NULL value of the service instance identifier shall be represented by a string of three asterisks (‘***’). This value shall not be accepted for input.

3.5.4.2 After creation, the value of the service instance identifier shall be NULL; i.e., the distinguished name shall not contain any components.

3.5.5 CREDENTIALS

3.5.5.1 Objects of the credentials class shall store the following attributes, as defined by reference [18] for the simple authentication scheme:

a) the time when the credentials have been created;

b) a random number;

c) a message digest produced according to the procedure defined in 3.5.6.4.

3.5.5.2 The credentials class shall provide read and write access to its attributes by the interface `ISLE_Credentials`.

3.5.6 SECURITY ATTRIBUTES

3.5.6.1 Objects of the class handling security attributes shall store the following attributes:

a) User name. The following rules shall apply for this attribute:

   1) the user name is a character string of 3 to 16 characters; and

   2) the user name must be identical to the authority identifier of the application by which the application is identified in the BIND invocation and the BIND return.

b) Password. The following rules shall apply for this attribute:

   1) the password is an octet string of 6 to 16 octets; and

   2) SLE API components make no assumptions on the contents of the octets and use the octet string as supplied.

3.5.6.2 Objects handling security attributes shall not check the length of the user name and the password but rely on the client supplying the attributes to pass strings of the correct length.

NOTE – It is expected that the components API Proxy and API Service Element check the length of the user name and password when reading the configuration database.

3.5.6.3 The services of the class shall be made available by the interface `ISLE_SecAttributes`. The features provided shall include:

a) write access to the attributes stored by the object;

NOTE – Because objects hold sensitive information, the interface shall not support read access.

b) test of two objects of the class for equality;

c) generation of credentials from the security attributes stored; and

d) authentication of credentials using the security attributes stored.

3.5.6.4 Generation of credentials shall be performed according to the protected simple authentication procedure (Protected 1) defined in reference [18] and detailed by the following specifications.

3.5.6.4.1 The following information shall be encoded using the ASN.1 syntax defined in reference [15] and the Distinguished Encoding Rules (DER) specified in reference [16]:

NOTE – Encoding the information with DER provides a platform independent bit pattern from which a hash code can be generated. Use of ASN.1 and DER for generation of credentials does not imply that ASN.1 or DER is used for encoding of data exchanged between the service user and the service provider. Given the simplicity of the ASN.1 type, encoding can be easily handcrafted and use of an ASN.1 compiler is not required.

a) the current time, using the CCSDS day segmented time code without the P-field;

b) a random number generated by the class;

c) the user name stored in the object; and

d) the password stored in the object.

3.5.6.4.2 The ASN.1 type used for encoding shall be defined as

```
HashInput ::= SEQUENCE
{ time          OCTET STRING (SIZE(8))
, randomNumber  INTEGER (0 .. 2147483647)
, userName      VisibleString
, passWord      OCTET STRING
}
```

3.5.6.4.3 The output of the encoder shall be passed through a one-way hash function to obtain a message digest.

3.5.6.4.4 A new credentials object shall be created and user name, the random number, the time, and the message digest shall be passed to that object.

3.5.6.5 Authentication of credentials shall be performed according to the protected simple authentication procedure (Protected 1) defined in reference [18] and detailed by the following specifications.

3.5.6.5.1 The time in the credentials shall be checked against the current time. If the time difference is larger than the acceptable delay passed as an argument, authentication shall fail.

3.5.6.5.2 The following information shall be encoded using the ASN.1 type defined in **Error! Reference source not found.** and the Distinguished Encoding Rules:

a) the time obtained from the credentials, in the CCSDS format;

b) the random number obtained from the credentials;

c) the user name stored in the object; and

d) the password stored in the object.

3.5.6.5.3 The string shall be passed through a one-way hash function to obtain a message digest.

3.5.6.5.4 The message digest shall be compared with the message digest in the credentials. If these match, authentication shall be regarded as successful. Otherwise, authentication shall fail.

3.5.6.6 The one-way hash function used is SHA-1 defined by reference [21].

3.5.7 COMPONENT OBJECTS AND INTERFACES

The component SLE Utilities shall implement the following component objects and interfaces:

NOTE – Component objects are defined in annex D of this specification. As explained there, a component object is an externally visible entity that may be implemented by a single object or by several internal objects, which co-operate to provide the required external view. As specified in annex D, every component object shall support the interface `IUnknown` in addition to the interfaces listed in this subsection. The interfaces referenced in the following are specified in annex A.

a) an object for the utility factory exporting the interface `ISLE_UtilFactory`;

b) objects for the time utility exporting the interface `ISLE_Time`;

c) objects for the service instance identifier exporting the interface `ISLE_SII`;

d) objects for the credentials exporting the interface `ISLE_Credentials`; and

e) objects for the security attributes exporting the interface `ISLE_SecAttributes`.

NOTE – Separate utility objects shall be supported for every object created via a call to the utility factory.

3.6 SLE APPLICATION

3.6.1 OBLIGATIONS

NOTE – An application using the SLE API must implement a set of interfaces defined by the API and perform specific tasks required for correct functioning of the API. This subsection summarizes the obligations of the application.

3.6.1.1 The application shall create all API components and configure them, see 3.6.5.

3.6.1.2 The application shall control processing of the service element and participate in service instance management, see 3.6.6.

3.6.1.3 The application shall implement and export the interface `ISLE_ServiceInform` used by the service element for passing of SLE protocol data units received from the peer SLE application for one service instance.

3.6.1.4 The application shall implement and export an interface by which API components can enter records to the system log and notify the application of specific events, see 3.6.2.

3.6.1.5 The application shall implement and export an interface by which API components can enter event trace records for diagnostic purposes. The application shall also start and stop tracing using the interface `ISLE_TraceControl` exported by the components. See 3.6.3.

3.6.1.6 The application may implement and export an interface by which the API component 'SLE Utilities' can obtain current time, see 3.6.4.

NOTE – An application using this feature may provide simulated time to the API.

3.6.1.7 The application shall participate in memory management by applying the reference counting scheme for component interfaces specified in annex D to this specification and using the API memory manager via the interface `IMalloc`.

NOTE – API memory management is specified in 3.5.2.

3.6.1.8 The application shall terminate processing of the API and control orderly shutdown of the API, see 3.6.5.

3.6.2 LOGGING AND NOTIFICATION

NOTES

1 The specifications in this subsection partially apply to clients of the interface `ISLE_Reporter`.

2 API components shall apply the following guidelines for production of log records and notifications. Errors detected by a component shall always be logged, providing as much information as possible to support investigation of the problem. Errors reported by a lower layer component shall not be logged unless important information can be added. Nominal events shall only be logged when the higher layer component or the application is not informed of the event by other means.

3 Notifications shall be constrained to events related to security, failure of the communication system, and to events that cannot be detected by the application by

other means. The event shall be notified by the highest layer in the API that can detect the event.

3.6.2.1 The SLE Application shall export the interface `ISLE_Reporter` by which API Proxy and the API Service Element can enter log records and notify the application of specific alarms.

NOTE – An application interface conforming to this specification must provide the interface `ISLE_Reporter`. However, this specification does not prescribe how the application handles the information passed to the interface.

3.6.2.2 The implementation of the interface `ISLE_Reporter` shall be multi-thread safe.

3.6.2.3 Log records shall be classified as 'alarms' or 'information messages'.

3.6.2.3.1 Alarms shall be raised for non-nominal events, including but not limited to:

a) security alarms (access violations and authentication failures);

b) communication system failures;

c) incorrect specification of operation parameters;

d) protocol errors;

e) configuration deficiencies; and

f) errors that might be caused by a malfunctioning component.

3.6.2.3.2 Information messages shall report nominal events for documentation purposes.

3.6.2.4 A log record shall be an ASCII string without any formatting characters.

NOTE – This specification does not define a maximum length for the string. Implementers should consider that most applications impose constraints on the length of log records and might have to truncate long strings. Therefore, the message should be kept as short as possible.

3.6.2.5 For every log record the following additional information shall be supplied, using the arguments of the method `LogRecord()`:

a) the identification of the component that produced the log record;

b) the service instance identifier, if applicable;

c) the classification of the log record as defined in 3.6.2.3; and

d) a unique identification number, which is referenced in the documentation supplied by the implementation of the API component.

NOTE – The time of the event is not passed as an argument. It is expected that the time be added by the method `LogRecord()`.

3.6.2.6 The application shall be notified of the following types of events using the function `Notify()`:

a) access violation alarms;

b) authentication failures;

c) communication system failures;

d) premature termination of an association by a component before the higher layer becomes aware of association establishment; and

e) premature termination of an association by the peer system before the higher layer becomes aware of association establishment.

3.6.2.7 The following information shall be supplied with a notification, using the arguments of the method `Notify()`:

a) the type of the notification as defined in 3.6.2.6;

b) the identification of the component that issued the notification;

c) the service instance identifier, if applicable;

d) a unique identification number, which is referenced in the documentation supplied by the implementation of the API component; and

e) optionally an additional text with a maximum length of 20 characters.

NOTE – The time of the event shall not be passed as an argument. It is expected that the time be added by the method `Notify()`.

3.6.3 DIAGNOSTIC TRACES

NOTES

1 The following specifications partially apply to clients of the interface `ISLE_Trace`.

2 This specification does not prescribe how an application deals with the trace records passed to the interface `ISLE_Trace`.

3 This specification does not prescribe how a trace record is formatted. When defining the layout, it should be considered that traces are generally used for printout and must be readable for humans. The output should be understandable to engineers that are not programmers.

3.6.3.1 Components supporting traces of events shall generate trace records that are passed to the interface `ISLE_Trace`, provided by the SLE Application.

NOTE – Components supporting traces shall implement the interface `ISLE_TraceControl`. If a component does not support tracing, a query for that interface shall be rejected.

3.6.3.1.1 The implementation of the interface `ISLE_Trace` shall be multi-thread safe.

3.6.3.1.2 The events for which for which trace records are generated and the amount of information that is entered in trace records shall be controlled by a trace level argument to the method `StartTrace()` in the interface `ISLE_TraceControl`. The trace levels are defined as follows:

a) 'Low' – state changes are traced. The information includes the old state, the new state, and the event that caused the state change.

b) 'Medium' – the trace additionally includes the type of all PDUs processed as well as additional interactions between components.

NOTE – An example for an additional interaction is the report by the proxy that a PDU has been transmitted. Further local events may be added by an implementation.

c) 'High' – the trace additionally contains a printout of all parameters of the PDU processed. The maximum length for the printout of one argument is constrained by the associated argument of the method StartTrace().

d) 'Full' – the trace additionally contains a dump of the encoded data sent to and received from the network.

3.6.3.1.3 For every trace record the following additional information shall be supplied, using the arguments of the method `TraceRecord()` defined in `ISLE_Trace`:

a) the identification of the component that produced the trace record; and

b) the service instance identifier, if applicable.

NOTE – The time of the event is not required as it can be added by the method `TraceRecord()`.

3.6.4 TIME SOURCE

NOTES

1 Applications may require that API components use a time source supplied by the application. Such a feature might be needed if the application uses an external time source, which is not necessarily synchronized with system time. It might also be used within simulation campaigns where a simulation might have to run in 'future time' without changing system time.

2 The Time Source interface specified in this subsection is used by the API component 'SLE Utilities' to set current time in the interface `ISLE_Time`. As API components are required to use that interface for handling of time, the time reference is distributed to all other API components. If the application opts not to use this feature, the component 'SLE Utilities' uses system time.

3.6.4.1 Applications wishing to provide a time source to the API shall export the interface `ISLE_TimeSource` and pass this interface to the creator function of the API component 'SLE Utilities'.

3.6.4.2 The interface `ISLE_TimeSource` shall provide a method returning the current time in CCSDS CDS format.

3.6.4.3 The time returned by the interface `ISLE_TimeSource` may have a positive or negative offset to the system time. However, API components may rely on the fact that this offset remains constant within the limits of the accuracy for timers defined in this specification.

NOTES

1 The required timer accuracy is specified in 3.7.2.8.

2 A constant offset from system time allows implementation of timers using standard services of the operating system.

3.6.5 INITIALIZATION AND SHUTDOWN OF THE API

3.6.5.1 The application shall create the API components needed for the specific installation using the creator functions provided by the components.

NOTE – The sequence in which the components must be created is partially determined by information required by the creator function. For instance, the creator function for operation objects requires a pointer to the utility factory.

3.6.5.2 The application shall configure and link the API components.

3.6.5.2.1 The application shall configure the service element providing it the path name of the configuration file and the required interface references.

NOTE – Configuration of the service element is specified in 3.3.10.

3.6.5.2.2 The application shall configure all proxy instances with the path name of the configuration file for each proxy and the required interface references including the reference to the interface `ISLE_Locator` obtained from the service element.

NOTE – Configuration of the proxy is specified in 3.2.12.

3.6.5.2.3 The application shall register each of the proxies with the service element.

NOTE – Registration of proxies with the service element is specified in 3.3.10.

3.6.5.3 After configuration of all components, the application shall start processing of the service element by invocation of the start method of the control interface selected according to 3.6.6.1.

NOTE – Processing of the proxies shall be started by the service element.

3.6.5.4 For closedown of the API, the application shall perform the following steps in the sequence specified.

3.6.5.4.1 The application shall call the terminate method of the selected control interface on the service element.

NOTE – Processing of the proxies shall be stopped by the service element.

3.6.5.4.2 The application shall release all API interfaces on which it holds references.

3.6.5.4.3 The application shall call the method `ShutDown()` of the administrative interfaces of the service element and of all proxies.

3.6.6 CONTROL OF THE SERVICE ELEMENT

3.6.6.1 For the interaction with the service element, the application shall provide one of the behaviors defined in 3.7 and control the service element using the interface associated with that behavior.

3.6.6.1.1 If the application provides the sequential behavior, it shall also provide the event monitor and the timer handler defined in 3.7.2.

3.6.6.2 The application shall create and delete service instances using the interface `ISLE_SIFactory` provided by the service element.

3.6.6.3 The application shall configure service instances providing the common service parameters defined in this specification and the service-type specific parameters defined in the relevant supplemental Recommended Practice for the service-specific API.

3.6.6.4 A service provider application shall update service instances with values of parameters that are modified by the service production process.

3.6.7 COMPONENT OBJECTS AND INTERFACES

3.6.7.1 An SLE Application shall implement the following component objects and interfaces:

NOTE – Component objects are defined in annex D of this specification. As explained there, a component object is an externally visible entity that may be implemented by a single object or by several internal objects, which co-operate to provide the required external view. As specified in annex D, every component object shall support the interface `IUnknown` in addition to the interfaces listed in this subsection. The interfaces referenced in the following are specified in annex A.

a) objects for use by service instances exporting the interface `ISLE_ServiceInform`;

NOTE – A separate component object shall be provided for every service instance created by the application.

b) one or more objects accepting log records and notifications and exporting the interface ISLE_Reporter;

c) one or more objects exporting the interface ISLE_EventMonitor, if the application requires the interface behavior ‘sequential’;

d) one or more objects exporting the interface ISLE_TimerHandler if the application requires the interface behavior ‘sequential’.

3.7 HANDLING OF IN PROCESS THREADS AND EXTERNAL EVENTS

3.7.1 GENERAL SPECIFICATIONS

NOTES

1 In order to ensure substitutability, handling of threads (or other implementations of concurrent flows of control) must be well defined at interfaces between components. This specification defines a single threaded (sequential) and a multi-threaded (concurrent) option. For the single threaded option, it also defines an interface by which the client offers means for components to wait for external events. For the multi-threaded option, components are expected to handle external events internally. Components are required to support one of these options but may support both. The terminology and the concepts are explained in 3.3.4.

2 The specifications in this subsection apply to active API components, i.e., the API Service Element and the API Proxy, as well as to the interfaces provided by the SLE

Application. Relevant specifications for the components ‘SLE Operations’ and ‘SLE Utilities’ are defined in 3.4 and 3.5.

3.7.1.1 API components shall provide one of the following behaviors for interfaces:

a) sequential behavior, in which methods of the interface must be invoked sequentially by different flows of control;

NOTE – Sequential behavior and the associated control interface are defined in 3.7.2.

b) concurrent behavior in which methods of an interface may be invoked concurrently by different flows of control.

NOTE – Concurrent behavior and the associated control interface are defined in 3.7.3.

3.7.1.2 A component shall provide the same behavior on all interfaces provided to one client and expect the same behavior for the complementary interfaces provided by the client.

3.7.1.3 API components shall support a control interface according to the behavior provided on their interfaces. This control interface shall provide methods to start and terminate processing of the component.

3.7.2 SEQUENTIAL BEHAVIOR

3.7.2.1 A component providing sequential interface behavior shall be controlled by the interface `ISLE_Sequential` exported by the component.

3.7.2.2 Processing of the component shall be started by the method `StartSequential()`, which shall pass references to the interfaces `ISLE_EventMonitor` and `ISLE_TimerHandler` as arguments.

NOTE – The event monitor and its use by the component are defined in 3.7.2.7. The timer handler and its use by the component are specified in 3.7.2.8.

3.7.2.3 The method `StartSequential()` shall return as soon as processing of the component has started.

3.7.2.4 The component shall guarantee that calls to complementary interfaces provided by its client are performed in the thread of control that originates from:

a) a call of the client to one of the interfaces exported by the component;

b) a call to the method `ProcessEvent()` in the interface `ISLE_EventProcessor` passed to the event monitor; or

c) a call to the method `ProcessTimeout()` in the interface `ISLE_TimeoutProcessor` passed to the timer handler.

NOTE – Use of multiple threads within the component is not excluded. However, the component must ensure that no thread created within the component or in a component other than the client enters client code.

3.7.2.5 The client shall guarantee that all calls to component interfaces are performed in a single thread of control at a time.

NOTE – Use of multiple threads by the client is not excluded, but the client must ensure that calls to the component interfaces are strictly serialized.

3.7.2.6 For interfaces with sequential behavior, sequence counting for transfer of SLE protocol data units as defined in 3.7.3.5 is not required. The sequence count argument in the associated methods shall be set to zero.

3.7.2.7 The event monitor shall provide a service to the component to wait for external events.

3.7.2.7.1 An event, which the event monitor should wait for, shall be registered with the method `AddEvent()` passing an event handle and a reference to the interface `ISLE_EventProcessor`.

NOTE – When events make use of UNIX file descriptors and event types (see annex A) `AddEvent()` must be called separately for read events, write events, and exceptions on a file descriptor, if the event monitor shall report these events.

3.7.2.7.2 An event shall be de-registered with the method `RemoveEvent()`, which references the event handle that shall be removed from the list of monitored events.

3.7.2.7.3 The event monitor shall support waiting for several events in parallel. If the event monitor constrains the number of events that can be registered, it shall return an error code indicating ‘overflow’ when this number is exceeded.

NOTE – It is noted that an event monitor with too restrictive constraints can prevent proper operation of the component.

3.7.2.7.4 When the event monitor detects an event, it shall call the method `ProcessEvent()` on the interface that has been registered for that event.

3.7.2.7.5 When the event monitor is no longer able to monitor an event for whatever reason, it shall remove the event and inform the event processor using the method `MonitorAbort()`.

NOTE – If more than one event must be removed, the method shall be invoked for every event.

3.7.2.8 The timer handler shall provide a service to the component to have timers started and be informed when the timer expires.

3.7.2.8.1 A timer shall be started by the method `StartTimer()` passing the timeout value and a reference to the interface `ISLE_TimeoutProcessor`. When the timer has been started, the method shall provide a timer identifier for later reference.

3.7.2.8.2 The timer handler shall allow specifying the timeout value with a resolution of one second.

3.7.2.8.3 The timer handler shall support several running timers in parallel. If the timer handler constrains the number of timers that can be running, it shall return an error code indicating 'overflow' when this number is exceeded.

NOTE – It is noted that timer handler with too restrictive constraints can prevent proper operation of the component.

3.7.2.8.4 When the timer expires, the timer handler shall call the method `ProcessTimeout()` of the interface `ISLE_TimeoutProcessor`, which has been registered with the method `StartTimer()`.

3.7.2.8.5 The timer handler shall provide the method `CancelTimer()` of the interface `ISLE_TimerHandler`, with which an active timer can be cancelled.

3.7.2.8.6 The timer handler shall provide the method `RestartTimer()` of the interface `ISLE_TimerHandler`, with which an active timer can be cancelled and restarted with a new timeout value.

3.7.2.8.7 As an option, the timer handler shall support an invocation identifier to be associated with the activation of a timer.

3.7.2.8.8 The invocation identifier shall be passed as an optional argument to the method `StartTimer()` or `RestartTimer()` of the interface `ISLE_TimerHandler`.

3.7.2.8.9 The timer handler shall memorize the identifier and pass it to the call of the method `ProcessTimeout()` in the interface `ISLE_TimeoutProcessor` when the timer expires.

NOTE – The invocation identifier supports handling of race conditions in a multi-threaded environment. If a timer is restarted just before it expires, a call to the method `ProcessTimeout()` in the interface `ISLE_TimeoutProcessor` can actually result from a previous call to `StartTimer()`. Such race conditions cannot be avoided, but unwanted calls to `ProcessTimeout()` can be identified and ignored, as the causality of the call can be determined using the invocation identifier.

3.7.2.8.10 When the timer handler is no longer able to process an active timer for whatever reason, it shall cancel the timer and inform the timeout processor using the method `HandlerAbort()`.

NOTE – If more than one timer must be aborted, the method shall be invoked for every timer.

3.7.2.9 Processing of the component shall be terminated by calling the method `TerminateSequential()` of the interface `ISLE_Sequential`. The method shall ensure that all events registered by the component are removed from the event monitor and all running timers have been cancelled.

3.7.3 CONCURRENT BEHAVIOR

3.7.3.1 A component providing concurrent interface behavior shall be controlled by the interface `ISLE_Concurrent` exported by the component.

3.7.3.2 Processing of the component shall be started by the method `StartConcurrent()`, which shall return as soon as processing has started.

3.7.3.3 The component must expect methods in an interface exported to the client to be called concurrently by separate threads of control.

3.7.3.4 The client of the component must expect methods in an interface passed to the client to be called concurrently by separate threads of control.

3.7.3.5 In order to support sequence preservation for SLE protocol data units, methods passing PDUs across an interface with concurrent behavior shall support sequence counting.

NOTES

1 The sequence count refers to the sequence in which PDUs have been received from the network or have been supplied by the application. It is required in a multi-threaded environment, because the sequence is not preserved when different PDUs are processed by different threads. This specification requires components to support sequence counting also in those cases where the specific implementation would preserve the sequence of PDUs. This would be the case when a component uses a single thread for PDUs transferred in one direction.

2 It is stressed that sequence counting is local to a given interface. For the service element, sequence-counts on the proxy interface can differ from those on the application interface.

3.7.3.5.1 The sequence count is a 32 bit unsigned integer.

3.7.3.5.2 The sequence count for a BIND invocation or a BIND return transmitted for one association shall be set to one.

NOTE – This implies that sequence counts shall restart at one when an association has been terminated or aborted and a new BIND invocation is issued.

3.7.3.5.3 For subsequent PDUs, the sequence-count shall be incremented by one.

3.7.3.5.4 Recycling of the sequence count to zero shall be supported.

3.7.3.5.5 The receiving entity shall define a window in which it accepts sequence counts. When receiving a sequence count outside of this window for a PDU, which is not a PEER-ABORT invocation, it shall reject the PDU with an error code indicating 'sequence error'.

NOTE – It is not required but recommended that the window size be configurable.

3.7.3.6 Components providing the concurrent behavior shall handle external events internally without further support by the client.

3.7.3.7 Processing of the component shall be terminated by the method `TerminateConcurrent()` of the interface `ISLE_Concurrent`. The method shall ensure that all threads started by the component are stopped such that graceful termination of the process becomes possible.

4 STATE TABLES

4.1 INTRODUCTION

This section defines detailed state tables for the processing of associations in the API Proxy and service instances in the API Service Element, which are derived from the state tables in the CCSDS Recommended Standards for SLE transfer services. The state tables in this specification differ from those in the CCSDS Recommended Standards for SLE transfer services in the following aspects:

a) the state tables are applicable to all service types;

b) specific state tables are provided for the API Proxy and the API Service Element; and

c) state tables for the SLE service user are explicitly specified.

The API Proxy and the API Service Element do not implement all aspects defined by the state tables in the CCSDS Recommended Standards for SLE transfer services. In particular, detection of and reaction to events in the service production process must be implemented by the application. The behavior defined by the state tables in the CCSDS Recommended Standards for SLE transfer services is achieved by interaction of the state machines in the API Proxy, the API Service Element, and the SLE Application.

4.2 NOTATION

The notation used for the state tables is the one specified by UML for state diagrams (see reference [J6]). This notation has been slightly extended to adapt it to state tables. It is summarized below together with the extensions. Extensions are highlighted by underlining. For formulation of conditions, the Object Constraint Language (OCL) specified by UML is used (see reference [J6]).

An incoming event in the event column is defined by

<origin> ':' <event-name> ['('<arguments>')']

Processing of the event is described by the following sequence

[<guard-condition>] [<action-expression>]* [<send-clause>]* [<state-transition>]

<guard-condition>	::=	'[' <condition> ']'
<condition>	::=	conditional expression formulated in OCL
<action-expression>	::=	'/' <action-name> ['(' <arguments> ')']
<send-clause>	::=	'^' <target> '.' <event-name> ['(' <arguments> ')']
<state-transition>	::=	'→' <new-state>

Transition to self is not shown in the tables.

In extension of the UML notation actions can be simple actions or compound actions. Compound actions are displayed in capital letters and are expanded using simple pseudo-code (IF, THEN, ELSE, END IF) together with the notational elements shown above.

For a detailed description of the syntax and an explanation of how it is to be interpreted, the UML specification should be consulted (see reference [J6]).

4.3 GENERAL ERROR HANDLING CONVENTIONS

For events received from another component, the following general rules are applied if the event is illegal in the current state:

a) If violation is due to misbehavior of the sending component the state shall not be changed and the request shall be rejected using a return code that indicates an error. In addition, the event shall be entered to the system log. When receiving such a return code, the invoking component is expected to abort the association and to provide as much information as possible to support investigation and correction of the problem.

b) In cases where the invoking component may not yet have become aware of a state change, the request shall be rejected by returning a code indicating that the state has changed. For instance, the service element may not yet have seen an UNBIND invocation when sending an invocation PDU. In this case, the proxy shall respond to the request with a return code indicating 'unbind pending'. This shall not be considered an error. The invoking component is expected to check the return code and adjust its state accordingly.

c) If the protocol error is due to a problem in the peer system, and it is not the responsibility of the sending component to check and handle the condition, the receiving component shall either generate and send a return with a negative result and the appropriate diagnostic, or abort the association. In this case, the state shall be adjusted and the method used to send the event returns with a code indicating success.

4.4 STATE TABLE FOR ASSOCIATIONS

4.4.1 PROCESSING CONTEXT

4.4.1.1 Overview

The presentation of the state table is based on the model described in section 2. The processing context used for specification of the state table is shown in figure 4-1. The proxy shall receive events either from the network interface (NIF) or the client interface (CIF) and send events to both the network interface and the client interface. The client interface shall include the interface `ISLE_SrvProxyInitiate`, which is exported by the proxy component and `ISLE_SrvProxyInform`, which is exported by the service element component.

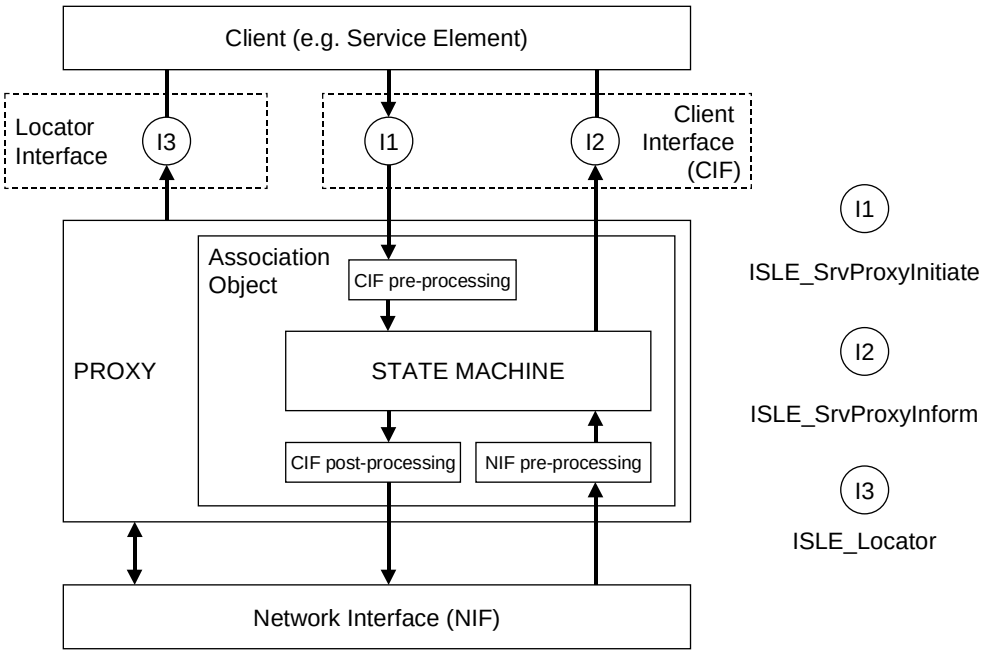

Figure 4-1: Processing Context for the Association State Table

When a BIND invocation is received from the network interface, the proxy shall create an association object of the correct service type and request the locator interface (`ISLE_Locator`) to locate a service instance. The locator interface shall return a reference to the interface `ISLE_SrvProxyInform`, if possible.

In order to simplify the state tables, processing steps that are common for all PDUs passed across an interface, and independent of the state of the association, have been excluded from the state tables. These are allocated to 'pre-processing' and 'post-processing' functions.

As shown in figure 4-1, it is assumed that pre-processing shall be performed on an event before it is passed to the state machine. If pre-processing fails, the associated action shall be performed as part of the pre-processing tasks and the event shall not be forwarded to the state machine. Post-processing of an event shall be performed after the event has been processed by the state machine. If a pre-processing or post-processing function encounters a situation in which the association must be aborted, it shall generate an internal event (INT: PeerAbort (reason)), which shall then be processed by the state machine.

Pre-processing tasks are defined for events received from the client interface and for events received from the network interface. Post-processing is only defined for events received from the client interface.

It is stressed that the specification of pre- and post-processing functions has only the purpose of simplifying the presentation of the state table. They do not prescribe any specific implementation. The only requirement on implementations is that the behavior defined by combination of the state table and the auxiliary functions shall actually be provided by the proxy.

4.4.1.2 Pre-Processing of Events Received from the Network Interface

Pre-processing of events received from the network interface includes the following tasks:

a) Decoding of the PDU. If decoding fails, the association shall be aborted.

b) Authentication of the peer identity, if authentication is required for the PDU. If authentication fails, the PDU shall be ignored and not passed to the state machine. 3.2.6.3.2 defines optional exceptions from this rule. These exceptions are not considered in this subsection.

c) Verification that the PDU is supported for the service type handled by the association. If that is not the case, the association shall be aborted with the diagnostic 'encoding error'.

d) For operation returns, retrieval of the operation object, which holds the associated operation invocation, using the invocation identifier. If the associated invocation cannot be found, the association shall be aborted with the diagnostic 'unsolicited invocation identifier'. If the operation object is located, it shall be removed from the list of pending returns.

4.4.1.3 Pre-Processing of Events Received from the Client Interface

Pre-processing of events received from the client interface includes the following tasks:

a) Verification that the PDU passed with the event is supported for the service type handled by the association. If the check fails, the event shall be rejected with an error indicating 'unknown PDU'.

b) Verification that the PDU can be queued for transmission. If the queue is full, the event shall be rejected with an error indicating 'overflow'.

4.4.1.4 Post-Processing of Events Received from the Client Interface

Post-processing of events received from the client interface includes the following tasks:

a) For invocations of confirmed operations, adding the operation object to the list of pending returns.

b) Generation and insertion of the credentials if authentication is required for the PDU. For PDUs of the BIND operation, insertion of the local application identifier. This step is omitted in the special 'pass-through' mode as specified in 3.2.7.

c) Encoding of the PDU.

d) Queuing of the PDU for transmission.

e) Transmission of the PDU, as soon as possible. If notification of transfer has been requested for that PDU, the task shall include the following steps:

1) if the PDU can be sent immediately, the event shall be acknowledged with a return code indicating ‘PDU transmitted’;

2) otherwise, the internal event ‘PDU transmitted’ shall be generated as soon as the PDU has been sent.

If the PDU is discarded on request of the client or because of an abort, the event ‘PDU Transmitted’ shall not be generated.

4.4.2 STATES

S1 - UNBOUND	An association in the initiator role has been created, but no BIND has been initiated yet or the association has been unbound or aborted. For an association in the responder role, the association object does not exist.
S2 - BIND PEND	A BIND invocation PDU has been processed successfully; the associated BIND return has not yet been received.
S3 - BOUND	The BIND operation has been completed successfully.
S4 - LOC UNBIND PEND	An UNBIND invocation issued by the local client has been processed; the peer proxy has not yet responded.
S5 - REM UNBIND PEND	An UNBIND invocation received from the peer proxy has been processed; the local client has not yet responded.

4.4.3 EVENTS

4.4.3.1 Events Received from the Client Interface (`ISLE_SrvProxyInitiate`)

BindInvoke	call to `InitiateOpInvoke()` with a BIND operation
BindReturn	call to `InitiateOpReturn()` with a BIND operation
UnbindInvoke	call to `InitiateOpInvoke()` with a UNBIND operation
UnbindReturn	call to `InitiateOpReturn()` with a UNBIND operation
PeerAbort	call to `InitiateOpInvoke()` with a PEER-ABORT operation
SrvPduInvoke	call to `InitiateOpInvoke()` with an operation that is valid for the given service type
SrvPduReturn	call to `InitiateOpReturn()` with an operation that is valid for the given service type
DiscardBuffer	call to `DiscardBuffer()`

4.4.3.2 Events Sent to the Client Interface (`ISLE_SrvProxyInform`)

BindInvoke	call to `InformOpInvoke()` with a BIND operation

BindReturn	call to `InformOpReturn()` with a BIND operation
UnbindInvoke	call to `InformOpInvoke()` with a UNBIND operation
UnbindReturn	call to `InformOpReturn()` with a UNBIND operation
PeerAbort	call to `InformOpInvoke()` with a PEER-ABORT operation
ProtocolAbort	call to `ProtocolAbort()`
SrvPduInvoke	call to `InformOpInvoke()` with an operation that is valid for the given service type
SrvPduReturn	call to `InformOpReturn()` with an operation that is valid for the given service type
PDUTransmitted	call to `PDUTransmitted()`

4.4.3.3 Events Sent to the Locator Interface (`ISLE_Locator`)

locateInstance	call to `LocateInstance()`

4.4.3.4 Events Received from the Network Interface

BindInvoke	reception of a BIND invocation PDU
BindReturn	reception of a BIND return PDU
UnbindInvoke	reception of a UNBIND invocation PDU
UnbindReturn	reception of a UNBIND return PDU
SrvPduInvoke	reception of a PDU with an invocation that is valid for the service type
SrvPduReturn	reception of a PDU with a return that is valid for the service type
PeerAbort	indication of a peer abort procedure initiated by the peer proxy
Communication failure	any indication from the local communication service provider of a communications failure or breakdown of the connection

4.4.3.5 Events Sent to the Network Interface

BindInvoke	BIND invocation PDU
BindReturn	BIND return PDU
UnbindInvoke	UNBIND invocation PDU
UnbindReturn	UNBIND return PDU
SrvPduInvoke	a PDU with an invocation that is valid for the service type
SrvPduReturn	a PDU with a return that is valid for the service type

4.4.3.6 Internal Events

PDU transmitted	A PDU for which notification of transfer has been requested has been transmitted.
Peer Abort	The need to abort the association has been detected by one of the pre-processing functions.

4.4.4 PREDICATES

role = initiator	The association initiates the BIND operation. This predicate is true for all associations that have been created on request of the client.
role = responder	The association responds to a BIND invocation. This predicate is true for all associations that have been created by the proxy because of an incoming BIND invocation.
result = positive	The result parameter in the PDU indicates 'positive result'.
result = negative	The result parameter in the PDU indicates 'negative result'.
instance located	The locator has returned an instance of the client interface.
id registered	The initiator identifier (user name) presented by a BIND invocation PDU or the responder identifier presented in the BIND return PDU is registered in the configuration database of the proxy.
responder = expected	The responder identifier in a BIND return PDU is the one expected. If the role of the association is 'initiator', the ID must be the one specified by the BIND operation object. If the role of the association is 'responder', it must match the local application identifier.
version supported	The version number presented in the BIND invocation is supported for the specified service type.
type supported	The service type presented in the BIND invocation PDU is supported.
bind arguments ok	The arguments of a BIND invocation issued by the local client match the definitions in the configuration database of the proxy. The expected responder identifier is registered.

4.4.5 ACTIONS

4.4.5.1 Discrete Actions

/abort connection(diagnostic)	All PDUs queued for transmission shall be discarded and the connection to the peer system terminated in an abortive manner, using the most efficient procedure available. The procedure applied must make sure that its effect can be interpreted by the peer as a PEER-ABORT and that the diagnostic is made available to the peer proxy.
/cleanup	All resources allocated by the association shall be released. In particular, all operation objects to which the association still holds a reference shall be released and the list of pending returns cleared.
/create association	Create a new association object in the role of a responder. In a strict sense, this action is performed before processing of the state table starts.
/delete association	The association object is deleted; following this action processing of the state table shall be assumed to cease. Therefore, no state change shall be indicated.

/prevent	If the technology is connection-oriented and a single connection is used for an association, the event cannot happen. If the event can occur, the proxy shall handle it in a manner that the operation of the API is not affected.
/reject(reason)	The function call returns with a result code indicating the reason.
/terminate connection	The connection to the peer system is released in an orderly manner.
/discard invocations	Remove all invocation PDUs from the send queue and discard them.
/discard buffers	Remove all TRANSFER-BUFFER PDUs from the send queue and discard them.

4.4.5.2 Compound Actions

/ABORT(diagnostic) is defined as

/abort connection(diagnostic)
^CIF.PeerAbort(diagnostic)
/cleanup
IF role = initiator THEN → S1
ELSE /delete association
END IF

/PROCESS BIND INV is defined as

```
IF not id registered THEN
   ^NIF.UnbindReturn('access denied')
   /cleanup /delete association
ELSE
   IF not type supported THEN
      ^NIF.UnbindReturn('type not supported')
      /cleanup /delete association
   ELSE
      IF not version supported THEN
         ^NIF.UnbindReturn('version not supported')
         /cleanup /delete association
      ELSE
         ^Locator.locateInstance
         IF not instance located THEN
                ^NIF.UnbindReturn(error returned by locateInstance)
                /cleanup /delete association
         ELSE
                ^CIF.BindInvoke
                → S2
         END IF
      END IF
   END IF
END IF
```

NOTE – If the initiator identifier is registered and authentication is required, authentication must be performed before processing starts. If authentication fails, the request shall be ignored.

/PROCESS BIND RET is defined as

```
IF not id registered THEN
   /ABORT('access denied')
   /cleanup →S1
ELSE
   IF not responder = expected THEN
      /ABORT('unexpected responder id')
      /cleanup →S1
   ELSE
      ^CIF.BindReturn
      IF result = positive THEN
            → S3
      ELSE
            /cleanup → S1
      END IF
   END IF
END IF
```

NOTE – If the responder identifier is registered and authentication is required, authentication must be performed before processing starts. If authentication fails, the request shall be ignored.

4.4.6 STATE TABLE FOR ASSOCIATIONS

	S1- UNBOUND {1}	S2 - BIND PEND	S3 - BOUND	S4 - LOC UNBIND PEND	S5 - REM UNBIND PEND
CIF: BindInvoke	[bind arguments ok] ^NIF.BindInvoke → S2 [not bind arguments ok] /reject(config error)	/reject(protocol error)			
CIF: BindReturn	/reject(protocol error)	[role = initiator] /reject(protocol error) [role = responder] ^NIF.BindReturn [result = positive] → S3 [result = negative] /terminate connection /cleanup /delete association ***{12}***	/reject(protocol error)		
CIF: UnbindInvoke	/reject(protocol error)		[role = initiator] ^NIF.UnbindInvoke → S4 [role = responder] /reject(protocol error)	/reject(protocol error)	
CIF: UnbindReturn	/reject(protocol error)			[role = intiator] /reject(protocl error) [role = responder] ^NIF:UnbindReturn /terminate connection /cleanup /delete association ***{12}***	

	S1- UNBOUND {1}	S2 - BIND PEND	S3 - BOUND	S4 - LOC UNBIND PEND	S5 - REM UNBIND PEND
CIF: PeerAbort	/reject(protocol error)	/abort connection(diagnostic) /cleanup [role = initiator] → S1 [role = responder] /delete association ***{12}***			
CIF: SrvPduInvoke	/reject(protocol error)		^NIF.SrvPduInvoke	/reject(protocol error)	/reject(unbind pending) ***{2}***
CIF: SrvPduReturn	/reject(protocol error)		^NIF.SrvPduReturn	/reject(protocol error)	^NIF.SrvPduReturn ***{3}***
CIF: DiscardBuffer	/reject(protocol error)		/discard buffers	/reject(protocol error)	/discard buffers
NIF: BindInvoke	create association /PROCESS BIND INV ***{4}***	/ABORT(protocol eror) ***{5}***			
NIF: BindReturn	/prevent	[role = initiator] /PROCESS BIND RET [role = responder] /ABORT(protocol error) ***{6}***	/ABORT(protocol error) ***{7}***		
NIF: UnbindInvoke	/prevent	/ABORT(protocol erorr)	[role = initiator] /ABORT(protocol error) [role = responder] /discard invocations ^CIF.UnbindInvoke → S5	/ABORT(protocol error)	
NIF: UnbindReturn	/prevent	/ABORT(protocol error)		[role = initiator] ^CIF.UnbindReturn /cleanup → S1 [role = responder] /ABORT(protocol error)	/ABORT(protocol error)

	S1- UNBOUND {1}	S2 - BIND PEND	S3 - BOUND	S4 - LOC UNBIND PEND	S5 - REM UNBIND PEND
NIF: PeerAbort	/prevent	^CIF.PeerAbort /cleanup [role = initiator] → S1 [role = responder] /delete associaton			
NIF: Communication failure	/prevent	^CIF.ProtocolAbort /cleanup [role = initiator] → S1 [role = responder] /delete associaton			
NIF: SrvPduInvoke	/prevent	/ABORT(protocol error)	^CIF.SrvPduInvoke	^CIF.SrvPduInvoke ***{8}***	/ABORT(protocol error) ***{9}***
NIF: SrvPduReturn	/prevent	/ABORT(protocol error)	^CIF.SrvPduReturn	^CIF.SrvPduReturn ***{10}***	/ABORT(protocol error) ***{11}***
INT: PDU Transmitted	not applicable	^CIF.PDUTransmitted			
INT: Peer Abort (reason)	not applicable	/ABORT(reason)			

NOTES

1 With exception of the event NIF: `BindInvoke` the events in the state UNBOUND can only occur for an association in the initiator role. An association in the responder role is created when the event NIF: `BindInvoke` has been received. If the BIND invocation is accepted, the state is changed to BIND PENDING. Otherwise the object is deleted again.

2 If UNBIND has been initiated by the peer, the local client must not transmit any further invocations. In a multi-threaded system, the client may not have seen the UNBIND yet. The special return code ‘unbind pending’ is meant to indicate to the client that this action is not a bug (protocol error) but cannot be accepted because the state has already changed.

3 If UNBIND has been initiated by the peer, the local client may transmit all pending returns before the UNBIND return.

4 The cell only contains an entry for the responder role. For the initiator role, it is assumed that proxy will not be listening for BIND invocations such that this event cannot occur in the state UNBOUND.

5 The event can only occur, if the technology and the implementation allow transmission of a BIND invocation on an established association. In this case it could theoretically happen that the BIND invocation contains an initiator identifier that differs from the one presented in the original BIND invocation. If such an event is possible, the proxy is expected to generate an access violation alarm in addition to the actions specified.

6 If the (illegal) BIND return received by an association in the responder role carries a responder identifier, that is not expected, the proxy generates an access violation alarm in addition to the action specified.

7 The event can only occur, if the technology and the implementation allow transmission of a BIND return on an established association. In this case, it could theoretically happen that the BIND return contains a responder identifier that differs from the one presented in the original BIND return. If such an event is possible, the proxy is expected to generate an access violation alarm in addition to the actions specified.

8 If UNBIND has been sent by the local client, the peer should no longer send any invocations. However, the peer may not yet have seen the UNBIND invocation. Therefore, invocations are passed on to local client, which should ignore them.

9 After sending an UNBIND invocation, the peer must not send any further invocations.

10 If UNBIND has been sent by the local client, the peer may send all pending returns before sending the UNBIND return.

11 After sending an UNBIND invocation, the peer must not send any returns.

12 Following deletion of the association object, processing of the state table is assumed to cease. Therefore, no state change is indicated.

4.5 STATE TABLES FOR SERVICE INSTANCES

4.5.1 INTRODUCTION

Processing of a service instance within the API Service Element is defined by the following state tables:

a) SLE Service Provider

   1) Common state table (see 4.5.3.2);

   2) Return link SLE services state table (see 4.5.3.3);

   3) Forward link SLE services state table (see 4.5.3.4).

b) SLE Service User

   1) Common state table (see 4.5.4.2);

   2) Return link SLE services state table (see 4.5.4.3);

   3) Forward link SLE services state table (see 4.5.4.4).

The common state tables define processing of all events that have identical processing requirements for return services and for forward services. The specific tables for return services and forward services specify processing of the remaining events. They must be understood as a supplement to the common tables.

The common state tables are applicable to all service types; the return link tables and the forward link tables are applicable for all return link services and all forward link services respectively. Because some service types only use a subset of the SLE operations defined, not all events defined in the tables can occur for those services, unless there are serious errors in the application or in the API Proxy. If such events are encountered, the service instance is expected to reject them with an appropriate error code. These actions are not specifically shown in the state table.

For some of the actions defined in the state tables processing is service-type specific, but the fact that the action must be performed is independent of the service type. Obviously, actions related to events that are not supported by a given service type, are not applicable for that service type.

4.5.2 PROCESSING CONTEXT

4.5.2.1 Overview

The state tables are based on the model presented in section 2. The processing context is used for specification of the state tables is shown in figure 4-2. A service instance shall receive events from the Proxy Interface (PIF) and the Application Interface (AIF) and send events to both the Proxy Interface and the Application Interface.

The Proxy Interface comprises the interfaces `ISLE_SrvProxyInitiate`, exported by the proxy for an association and the interface `ISLE_SrvProxyInform`, exported by the service element. The Application Interface includes the interface `ISLE_ServiceInform`, exported by the application, and `ISLE_ServiceInitiate`, exported by the service element.

In addition, the service instance shall receive updates for service parameters from the application via the service-type specific interface `I<SRV>_SIUpdate`, which is referred to as Management Interface (MIF) in the figure. These parameters are needed to respond to a GET-PARAMETER request and to generate status reports.

Finally the locator interface (`ISLE_Locator`), by which the service element is informed of an incoming BIND invocation, needs to be considered. This interface is not shown in the figure.

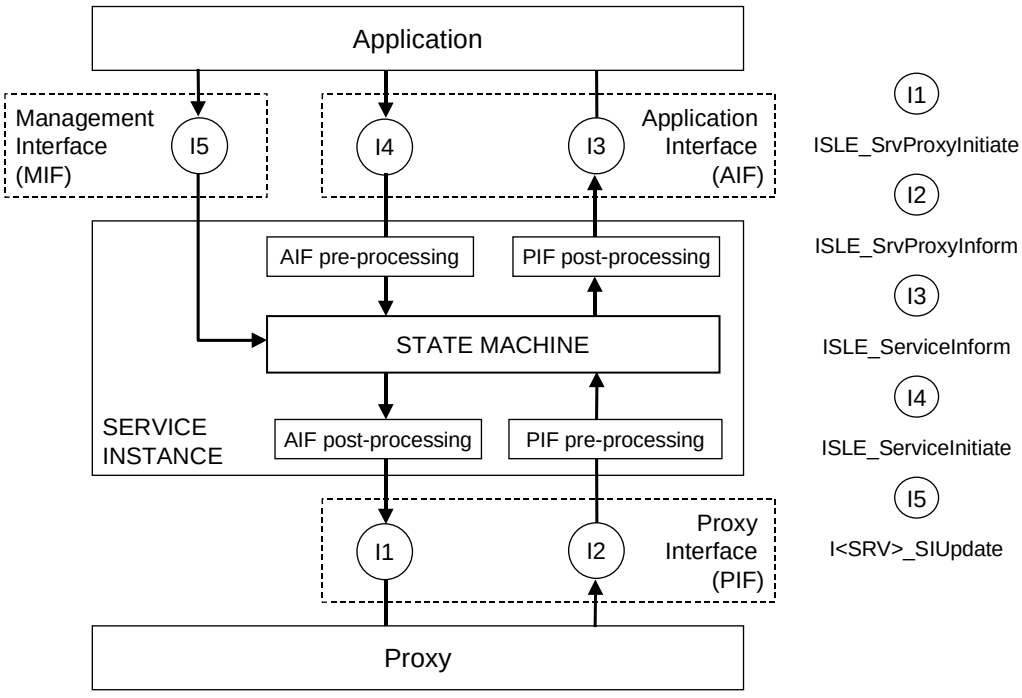

Figure 4-2: Processing Context for the Service Instance State Table

In order to simplify the state tables, processing steps that are common for all PDUs passed across an interface and are independent of the state of the service instance have been excluded from the state tables. These are allocated to 'pre-processing' and 'post-processing' functions.

As shown in figure 4-2, it is assumed that pre-processing is performed on an event before it is passed to the state machine. If pre-processing fails, the associated action shall be performed as part of the pre-processing function, and the event shall not be forwarded to the state machine. Post-processing of an event shall be performed after the event has been processed or generated by the state machine. If a pre-processing function or a post-processing function encounters a situation in which the association must be aborted, it shall generate an internal event (INT: Peer Abort(reason)), which shall then be processed by the state machine.

Furthermore, the state tables do not include processing performed by call to the locator interface. This processing is described to more detail in 4.5.2.6.

It is stressed that the specification of pre- and post-processing functions as well as the details related to calls on the locator interface have only the purpose of simplifying the presentation of the state tables. They do not prescribe any specific implementation. The only requirement on implementations is that the behavior defined by the combination of the state tables and the auxiliary functions shall actually be provided by service instance objects.

4.5.2.2 Pre-Processing of Events Received from the Proxy

Pre-processing of events received from the proxy includes the following tasks:

a) Verification that the PDU passed with the event is supported by the service type handled by the service instance. If that check fails, the event shall be rejected with the error 'unknown PDU'.

b) Verification that the PDU is compatible with the role of the service instance (SLE service user or SLE service provider). If the check fails, the association shall be aborted with the diagnostic 'protocol error'.

c) Checking that an operation object passing a return is actually on the list of pending remote returns. If that check fails, the request shall be rejected, because this problem should have been handled by the proxy. Otherwise the operation object shall be removed from the list of pending remote returns.

d) Canceling of the return timer for a return PDU.

e) Checking for duplicate invocation identifiers for confirmed invocations. If duplicate invocation identifiers are detected, the pre-processing function shall generate and send a return PDU with a negative result and the diagnostic 'duplicate invocation id'.

f) Checking of invocation and return arguments on completeness, consistency and range. If there is an error, the reaction depends on the type of the PDU. For a confirmed invocation, the pre-processing function shall generate and send a return

PDU with a negative result and the appropriate diagnostic. In all other cases it shall abort the association with the appropriate diagnostic code.

g) Checking of the consistency of the PDU and the parameters with the configuration of the service instance. If these checks fail, the function shall generate a return with a negative result and the appropriate diagnostic. These checks are service-type specific.

The checks and actions are partially service-type specific.

4.5.2.3 Post-Processing of Events Received from the Proxy

Post-processing of events received from the proxy includes the following tasks:

a) adding the operation object to the list of pending local returns for confirmed invocations;

b) passing of the PDU to the application.

These actions are independent of the service type.

4.5.2.4 Pre-Processing of Events Received from the Application

Pre-processing of events received from the application includes:

a) Checking that the PDU is valid for the service type handled by the service instance.

b) Verification that the PDU is compatible with the role of the service instance (SLE service user or SLE service provider).

c) Verification that an operation object used to forward a return PDU is on the list of pending local returns.

d) Checking of invocation and return arguments on completeness, consistency and range.

e) Checking of the consistency of the PDU and the parameters with the configuration of the service instance.

If there is any error, the pre-processing function shall reject the request with the appropriate return code. These tasks are partially service-type specific.

4.5.2.5 Post-Processing of Events Generated or Received from the Application

Post-processing of events received from the application includes:

a) Generation of a unique invocation identifier and inserting the id into the operation object, if the PDU transmitted is a confirmed invocation.

b) Addition of the operation object to the list of pending remote returns and starting of the return timer for confirmed invocations.

c) Forwarding the operation to the proxy for transmission. If the transfer request is rejected by the proxy, e.g., because the send queue is full, the association shall be aborted.

d) Because of the flow control mechanisms built into the API, queue overflow cannot be caused by transfer of space link data-units.

These tasks are not service-type specific.

4.5.2.6 Processing of Calls to the Locator Interface

Processing of calls to the locator interface includes the following steps:

a) Location of the service instance requested by the BIND invocation. If the service instance cannot be found, the function shall return with an error 'no such service instance'.

b) Verification that the initiator identifier matches the one defined for the service instance. If that is not the case, the function shall return with an error 'service instance not accessible to this initiator'.

c) Verification that the service instance is not already bound. If the service instance is bound, the function shall return with an error 'already bound'.

d) Verification that the scheduled provision period of the service instance has started and has not yet ended. If the check fails, the function shall return with the error 'invalid time'.

When receiving an error response from the locator, the proxy shall generate a BIND return with a negative result and the diagnostic related to the error code returned. If the locator returns a code indicating success and a pointer to the interface `ISLE_SrvProxyInform`, the proxy shall pass the BIND invocation to that interface. Except for location of the service instance, an implementation may choose to perform these checks by the service instance, when the BIND invocation has been passed to the interface `ISLE_SrvProxyInform.` If that is done, the service instance must generate the BIND return PDU.

4.5.3 PROVIDER SIDE STATE TABLES

4.5.3.1 States

All provider side state tables use the same set of states. The main states are identical to those defined in the CCSDS Recommended Standards for SLE transfer services. Sub-states have been added to allow presentation of further details related to the interactions with the application and the proxy. The states are defined as follows.

UNBOUND:UNBOUND	No user is bound.
UNBOUND:BIND PEND	A BIND invocation has been received, the application has not yet responded.
READY:BOUND	A BIND has been sent to the user and no START invocation has been received yet, or a STOP operation has been completed.
READY:START PEND	A START invocation has been received, the application has not yet responded.
READY:UNBIND PEND	An UNBIND invocation has been received, the application has not yet responded.
ACTIVE:ACTIVE	A START return with a positive result has been sent to the user.
ACTIVE:STOP PEND	A STOP invocation has been received, the application has not yet responded.

NOTE – Sub-states are only shown in the tables if needed. If the processing is identical for all sub-states, only the main state is entered in the table.

4.5.3.2 Common State Table—User Initiated Binding

4.5.3.2.1 Events

4.5.3.2.1.1 Events received from the Application Interface (`ISLE_ServiceInitiate`)

BindRet	call to `InitiateOpReturn()` with a BIND operation
UnbindRet	call to `InitiateOpReturn()` with a UNBIND operation
PeerAbortInv	call to `InitiateOpInvoke()` with a PEER-ABORT operation

4.5.3.2.1.2 Events sent to the Application Interface (`ISLE_ServiceInform`)

BindInv	call to `InformOpInvoke()` with a BIND operation
UnbindInv	call to `InformOpInvoke()` with a UNBIND operation
PeerAbortInv	call to `InformOpInvoke()` with a PEER-ABORT operation
ProtocolAbort	call to `ProtocolAbort()`
PPends	call to `ProvisionPeriodEnds()`

4.5.3.2.1.3 Events received from the Management Interface (`I<SRV>_SIUpdate`)

SetParameter	update of a service parameter

4.5.3.2.1.4 Events received from the Proxy Interface (`ISLE_SrvProxyInform`)

BindInv	call to `InformOpInvoke()` with a BIND operation
UnbindInv	call to `InformOpInvoke()` with a UNBIND operation
GetPrmInv	call to `InformOpInvoke()` with a GET-PARAMETER operation
ScheduleStatRepInv	call to `InformOpInvoke()` with a SCHEDULE-STATUS-REPORT operation
PeerAbortInv	call to `InformOpInvoke()` with a PEER-ABORT operation
ProtocolAbort	call to `ProtocolAbort()`

4.5.3.2.1.5 Events sent to the Proxy Interface (`ISLE_SrvProxyInitiate`)

BindRet	call to `InitiateOpReturn()` with a BIND operation
UnbindRet	call to `InitiateOpReturn()` with a UNBIND operation
GetPrmRet	call to `InitiateOpReturn()` with a GET-PARAMETER operation
ScheduleStatRepRet	call to `InitiateOpReturn()` with a SCHEDULE-STATUS-REPORT operation
StatusRepInv	call to `InitiateOpInvoke()` with a STATUS-REPORT operation
PeerAbortInv	call to `InitiateOpInvoke()` with a PEER-ABORT operation

4.5.3.2.1.6 Internal Events

Report timer expired	the periodic status report timer has expired
Return timeout	the time to wait for a specific return-PDU has elapsed
Provision period ends	the service instance provision period has ended
Peer Abort	peer abort event generated by a pre-processing function

4.5.3.2.2 Predicates

delivery mode = offline	The delivery mode of the service instance is ‘offline’.
report timer active	The periodic status report timer is active.
reason = end	The unbind-reason is ‘end of service provision’.
reason <> end	The unbind-reason is not equal ‘end of service provision’.
result = positive	The result parameter in the PDU indicates ‘positive result’.
result = negative	The result parameter in the PDU indicates ‘negative result’.
type = stop	The type parameter in the SCHEDULE-STATUS-REPORT invocation is set to ‘stop’.

type = periodically	The type parameter in the SCHEDULE-STATUS-REPORT invocation is set to ‘periodically’.

4.5.3.2.3 Actions

4.5.3.2.3.1 Discrete Actions

/reject(reason)	Reject the event by returning an error code to the function invoking the event.
/cancel report timer	Cancel the periodic status-report timer, if active.
/start report timer	Start the periodic status-report timer.
/generate end of PP	Generate the internal event ‘Provision period ends’.
/clear remote returns	Cancel return timers for all pending remote returns, clear the list of pending remote returns, and release operation objects.
/store parameter value	Store the value of the service parameter passed.

4.5.3.2.3.2 Compound Actions

/PROCESS SSREP(type) is defined as

```
IF delivery mode = offline THEN
   /reject(not supported in this delivery mode)
ELSE
   IF type = stop THEN
      IF report timer active THEN
         /cancel report timer
         ^PIF.ScheduleStatusRepRet(positive result)
      ELSE
         PIF.ScheduleStatusRepRet(already stopped)
      END IF
   ELSE
      /cancel report timer
      ^PIF.ScheduleStatusRepRet(positive result)
      ^PIF.StatusReportInv
      IF type = periodically THEN
         /start report timer
      END IF
   END IF
END IF
```

/ABORT(diagnostic)

Abort processing is forward/return-service specific, see /ABORT in 4.5.3.3 and 4.5.3.4.

/CLEANUP

Cleanup processing is forward/return-service specific, see /CLEANUP in 4.5.3.3 and 4.5.3.4.

4.5.3.2.4 Common State Table—Provider Side

	1 UNBOUND		2 READY			3 ACTIVE
	1.1 UNBOUND *{1}*	**1.2 BIND PEND**	**2.1 BOUND**	**2.2 START PEND**	**2.3 UNBIND PEND**	
PIF: BindInv	^AIF.BindInv → 1.2 ***{2}***	/reject(protocol error) ***{3}***				
AIF: BindRet	/reject(protocol error)	^PIF.BindRet [result = positive] → 2.1 [result =negative] → 1.1	/reject(protocol error)			
PIF: UnbindInv	/reject(protocol error)		/clear remote returns /cancel report timer ^AIF.UnbindInv → 2.3	/ABORT(protocol err) → 1.1	/reject(protocol error)	/ABORT(protocol error) → 1.1
AIF: UnbindRet	/reject(protocol error)				^PIF.UnbindRet /CLEANUP [reason = end] /generate end of PP → 1.1 [reason <> end] → 1.1	/reject(protocol error)
PIF: GetPrmInv	/reject(protocol error)		^PIF.GetPrmRet		/reject(protocol error)	^PIF.GetPrmRet
PIF: ScheduleStatRepInv	/reject(protocol error)		/PROCESS SSREP		/reject(protocol error)	/PROCESS SSREP
PIF: PeerAbortInv	/reject(protocol error)	^AIF.PeerAbortInv /CLEANUP → 1.1				
AIF: PeerAbortInv	/reject(protocol error)	^PIF.PeerAbortInv /CLEANUP → 1.1				

	1 UNBOUND		2 READY			3 ACTIVE
	1.1 UNBOUND *{1}*	**1.2 BIND PEND**	**2.1 BOUND**	**2.2 START PEND**	**2.3 UNBIND PEND**	
PIF: ProtocolAbort	/reject(protocol error)	^AIF.ProtocolAbort /CLEANUP → 1.1				
MIF: SetParameter	/store parameter value					
INT: PeerAbort(reason)	N/A	/ABORT(reason) → 1.1				
INT: Report timer expired	N/A	N/A	^PIF.StatusRepInv /start report timer		N/A ***{4}***	^PIF.StatusRepInv /start report timer
INT: Return timeout ***{5}***	N/A	N/A	/ABORT (return timeout) → 1.1		N/A ***{4}***	/ABORT(return timeout) → 1.1
INT: Provision period ends	^AIF.PPends	/ABORT(end of Provision Period) ^AIF.PPends → 1.1				

NOTES

1 In the state UNBOUND, events other than a BIND invocation can be received from the proxy only when the proxy fails to forward the initial BIND invocation.

2 All checks that need to be performed by the service element are performed by the method `LocateInstance()` defined by the Locator interface. If any of these checks fail, that function returns an error and the proxy responds with the associated BIND return.

3 The event can only occur when a BIND invocation is received on an established association, which must be prevented by the proxy. If a BIND invocation is received on a new association, the event must be passed to the locator, which will reject it with the error ‘already bound’.

4 This is N/A as the timer was cancelled when the UNBIND invocation arrived.

5 In this version of the Recommended Practice the provider never sends confirmed operations, so this event cannot happen if the API software is correctly implemented.

4.5.3.3 Return Link SLE Services

4.5.3.3.1 Events

4.5.3.3.1.1 Events received from the Application Interface (`ISLE_ServiceInitiate`)

StartRet	call to `InitiateOpReturn()` with a START operation
StopRet	call to `InitiateOpReturn()` with a STOP operation
TransferDataInv	call to `InitiateOpInvoke()` with a TRANSFER-DATA operation
SyncNotifyInv	call to `InitiateOpInvoke()` with a SYNC-NOTIFY operation

4.5.3.3.1.2 Events sent to the Application Interface (`ISLE_ServiceInform`)

StartInv	call to `InformOpInvoke()` with a START operation
StopInv	call to `InformOpInvoke()` with a STOP operation
ResumeDataTransfer	call to `ResumeDataTransfer()`
PeerAbortInv	call to `InformOpInvoke()` with a PEER-ABORT operation

4.5.3.3.1.3 Events received from the Proxy Interface (`ISLE_SrvProxyInform`)

StartInv	call to `InformOpInvoke()` with a START operation
StopInv	call to `InformOpInvoke()` with a STOP operation
PDUTransmitted	call to `PDUTransmitted()`

4.5.3.3.1.4 Events sent to the Proxy Interface (`ISLE_SrvProxyInitiate`)

StartRet	call to `InitiateOpReturn()` with a START operation
StopRet	call to `InitiateOpReturn()` with a STOP operation
TransferBufferInv	call to `InitiateOpInvoke()` with a TRANSFER-BUFFER operation. This event is always transmitted with the request to notify transmission of the PDU.
DiscardBuffer	call to `DiscardBuffer()`
PeerAbortInv	call to `InitiateOpInvoke()` with a PEER-ABORT operation

4.5.3.3.1.5 Internal Events

release timer expired	generated when the release timer expires

4.5.3.3.2 Predicates

result = positive	The result parameter in the PDU indicates ‘positive result’.
result = negative	The result parameter in the PDU indicates ‘negative result’.
timely online	The delivery mode is timely online.
complete online	The delivery mode is complete online.
online	The delivery mode is either timely online or complete online.
buffer full	The transfer buffer is full.
buffer queued	A transfer buffer has been passed to the proxy for transfer and the PDU Transmitted event has not yet been received for that buffer.
buffer empty	The transfer buffer is empty.
buffer discarded	The proxy has actually discarded the queued transfer buffer as indicated by the return code.
buffer transmitted	The transfer buffer passed to the proxy could be transmitted immediately as indicated by the return code of the function.
data transfer suspended	The application has been requested to suspend data transfer to the user.
end of data	The SYNC-NOTIFY invocation is ‘end of data’.

4.5.3.3.3 Actions

4.5.3.3.3.1 Discrete Actions

/reject(reason)	Reject the event by returning an error code to the function invoking the event.
/clear remote returns	Cancel return times for all pending remote returns, clear the list of pending remote returns, and release operation objects.
/clear local returns	Clear the list of pending local returns and release operation objects.
/cancel report timer	Cancel the periodic status-report timer, if active.
/reset service parameters	Reset the service parameters to the initial values. Resetting of service parameters must be checked individually for each parameter. Depending on the service type some parameters may have to be reset to the initial values, while others must keep their current values.
/start release timer	Start the release timer.
/cancel release timer	Cancel the release timer, if active.
/create new buffer	Create a new transfer buffer.
/append PDU	Append the PDU to the transfer buffer.
/prepend notification	Prepend the SYNC-NOTIFY invocation, indicating ‘data discarded due to excessive backlog’, to the transfer buffer.
/discard buffer	Discard transfer buffer and all contained PDUs.
/suspend data transfer	Request the application to suspend data transfer.

4.5.3.3.3.2 Compound Actions

/ABORT(diagnostic) is defined as

```
^PIF.PeerAbort(diagnostic)
^AIF.PeerAbort(diagnostic)
/CLEANUP
```

/CLEANUP is defined as

```
/clear remote returns
/clear local returns
/cancel release timer
/cancel report timer
/discard buffer
/set data transfer suspended = FALSE
/set buffer queued = FALSE
/reset service parameters
```

/BUFFER DATA is defined as

```
IF online and buffer empty THEN
   /start release timer
END IF
/append PDU
IF buffer full THEN
   IF buffer queued THEN
      IF timely online THEN
         ^PIF.DiscardBuffer
         IF buffer discarded THEN
            /prepend notification
         END IF
         ^PIF.TransferBuffer
         IF not buffer transmitted THEN
            /set buffer queued = TRUE
         END IF
         /cancel release timer
      ELSE
         IF complete online THEN
            /cancel release timer
         END IF
         /set data transfer suspended = TRUE
         /suspend data transfer
      END IF
   ELSE
      IF online THEN
```

```
            /cancel release timer
         END IF
         ^PIF.TransferBuffer
         IF not buffer transmitted THEN
            /set buffer queued = TRUE
         END IF
      END IF
      /create new buffer
   END IF
```

NOTE – Processing as specified here, uses a single transfer buffer. Multiple buffers can be used by an implementation to increase performance.

/PROCESS RELEASE TIMER is defined as

```
   IF buffer queued THEN
      IF timely online THEN
         ^PIF.DiscardBuffer
         IF buffer discarded THEN
            /prepend notification
         END IF
      END IF
      IF complete online THEN
         /suspend data transfer
      END IF
   END IF
   ^PIF.TransferBuffer
   IF not buffer transmitted THEN
      /set buffer queued = TRUE
   END IF
   /create new buffer
```

/PROCESS PDU TRANSMITTED is defined as

```
   /set buffer queued = FALSE
   IF data transfer suspended THEN
      /set data transfer suspended = FALSE
      ^AIF.ResumeDataTransfer
   END IF
```

/PROCESS STOP PDU is defined as

```
   IF not buffer empty THEN
      IF online THEN
         /cancel release timer
      END IF
      IF timely online THEN
```

```
            IF buffer queued THEN
               ^PIF.DiscardBuffer
               IF buffer discarded THEN
                  /prepend notification
               END IF
            END IF
         END IF
         ^PIF.TransferBuffer
         IF not buffer transmitted THEN
            /set buffer queued = TRUE
         END IF
      END IF
```

/PROCESS EOD is defined as

```
      /append PDU
      IF online THEN
         /cancel release timer
      END IF
      IF timely online THEN
         IF buffer queued THEN
            ^PIF.DiscardBuffer
            IF buffer discarded THEN
               /prepend notification
            END IF
         END IF
      END IF
      ^PIF.TransferBuffer
      IF not buffer transmitted THEN
         /set buffer queued = TRUE
      END IF
      /create new buffer
```

4.5.3.3.4 Return Link State Table—Provider Side

	1 UNBOUND	2 READY			3 ACTIVE	
		2.1 BOUND	2.2 START PEND	2.3 UNBIND PEND	3.1 ACTIVE	3.2 STOP PEND
PIF: StartInv	/reject(protocol err)	^AIF.StartInv → 2.2	/ABORT(protocol err) → 1.1	/reject(protocol error)	/ABORT(protocol error) → 1.1	
AIF: StartRet	/reject(protocol error)		^PIF.StartRet [result = positive] /create new buffer → 3.1 [result = negative] → 2.1	/reject(protocol error)		
PIF: StopInv	/reject(protocol err)	/ABORT(protocol error) → 1.1		/reject(protocol error)	^AIF.StopInv → 3.2	/ABORT (protocol error) → 1.1
AIF: StopRet	/reject(protocol error)					[result = positive] /PROCESS STOP PDU ^PIF.StopRet → 2.1 [result =negative] ^PIF.StopRet → 3.1
AIF: TransferDataInv	/reject(protocol error)				[data transfer suspended] /reject [not data transfer suspended] /BUFFER DATA	/reject(stop pending)
AIF: SyncNotifyInv	/reject(protocol error)				[data transfer suspended] reject(suspended) [not data transfer suspended] [end of data] /PROCESS EOD [not end of data] /BUFFER DATA	

	1 UNBOUND	2 READY			3 ACTIVE	
		2.1 BOUND	2.2 START PEND	2.3 UNBIND PEND	3.1 ACTIVE	3.2 STOP PEND
INT: Release timer expired	N/A				/PROCESS RELEASE TIMER	
PIF: PDUTransmitted	/reject	/PROCESS PDU TRANSMITTED				

4.5.3.4 Forward Link SLE Services

4.5.3.4.1 Events

4.5.3.4.1.1 Events received from the Application Interface (`ISLE_ServiceInitiate`)

StartRet	call to `InitiateOpReturn()` with a START operation
StopRet	call to `InitiateOpReturn()` with a STOP operation
TransferDataRet	call to `InitiateOpReturn()` with a TRANSFER-DATA operation
InvokeDirectiveRet	call to `InitiateOpReturn()` with an INVOKE-DIRECTIVE operation
AsyncNotifyInv	call to `InitiateOpInvoke()` with an ASYNC-NOTIFY operation
ThrowEventRet	call to `InitiateOpInvoke()` with a THROW-EVENT operation

4.5.3.4.1.2 Events sent to the Application Interface (`ISLE_ServiceInform`)

StartInv	call to `InformOpInvoke()` with a START operation
StopInv	call to `InformOpInvoke()` with a STOP operation
TransferDataInv	call to `InformOpInvoke()` with a TRANSFER-DATA operation
InvokeDirectiveInv	call to `InformOpInvoke()` with an INVOKE-DIRECTIVE operation
ThrowEventInv	call to `InformOpInvoke()` with a THROW-EVENT operation
PeerAbortInv	call to `InformOpInvoke()` with a PEER-ABORT operation

4.5.3.4.1.3 Events received from the Proxy Interface (`ISLE_SrvProxyInform`)

StartInv	call to `InformOpInvoke()` with a START operation
StopInv	call to `InformOpInvoke()` with a STOP operation
TransferDataInv	call to `InformOpInvoke()` with a TRANSFER-DATA operation
InvokeDirectiveInv	call to `InformOpInvoke()` with an INVOKE-DIRECTIVE operation
ThrowEventInv	call to `InformOpInvoke()` with a THROW-EVENT operation

4.5.3.4.1.4 Events sent to the Proxy Interface (`ISLE_SrvProxyInitiate`)

StartRet	call to `InitiateOpReturn()` with a START operation
StopRet	call to `InitiateOpReturn()` with a STOP operation
TransferDataRet	call to `InitiateOpReturn()` with a TRANSFER-DATA operation
InvokeDirectiveRet	call to `InitiateOpReturn()` with a INVOKE-DIRECTIVE operation
AsyncNotifyInv	call to `InitiateOpInvoke()` with an ASYNC-NOTIFY operation
ThrowEventRet	call to `InitiateOpReturn()` with a THROW-EVENT operation
PeerAbortInv	call to `InitiateOpInvoke()` with a PEER-ABORT operation

4.5.3.4.2 Predicates

result = positive	The result parameter in the PDU indicates ‘positive result’.
result = negative	The result parameter in the PDU indicates ‘negative result’.

4.5.3.4.3 Actions

4.5.3.4.3.1 Discrete Actions

/reject(reason)	Reject the event by returning an error code to the function invoking the event.
/clear remote returns	Cancel return timers for all pending remote returns, clear the list of pending remote returns, and release the operation objects.
/clear local returns	Clear the list of pending local returns and release the operation objects.
/cancel report timer	Cancel the periodic status-report timer, if active.
/reset service parameters	Reset the service parameters to the initial values. Resetting of service parameters must be checked individually for each parameter. Depending on the service type some parameters may have to be reset to the initial values, while others must keep their current values.

4.5.3.4.3.2 Compound Actions

/ABORT(diagnostic) is defined as

^PIF.PeerAbort(diagnostic)
^AIF.PeerAbort(diagnostic)
/CLEANUP

/CLEANUP is defined as

/clear remote returns
/clear local returns
/cancel report timer
/reset service parameters

4.5.3.4.4 Forward Link State Table—Provider Side

	1 UNBOUND	2 READY			3 ACTIVE	
		2.1 BOUND	2.2 START PEND	2.3 UNBIND PEND	3.1 ACTIVE	3.2 STOP PEND
PIF: StartInv	/reject(protocol error)	^AIF.StartInv → 2.2	/ABORT(protocol error) → 1.1	/reject(protocol error)	ABORT(protocol error) → 1.1	
AIF: StartRet	/reject(protocol error)		^PIF.StartRet [result = positive] → 3.1 [result = negative] → 2.1	/reject(protocol error)		
PIF: StopInv	/reject(protocol error)	/ABORT(protocol error) → 1.1		/reject(protocol error)	^AIF.StopInv → 3.2	/ABORT (protocol error) → 1.1
AIF: StopRet	/reject(protocol error)					^PIF.StopRet [result = positive] → 2.1 [result = negative] → 3.1
PIF: TransferDataInv	/reject(protocol error)	/ABORT(protocol error) → 1.1		/reject(protocol error)	^AIF.TransferDataInv → 3.1	/ABORT(protocol error) → 1.1
AIF: TransferDataRet	/reject(protocol error)				^PIF.TransferDataRet	
PIF: InvokeDirectiveInv	/reject(protocol error)	/ABORT(protocol error) → 1.1		/reject(protocol error)	^AIF.InvokeDirectiveInv	/ABORT (protocol error) → 1.1
AIF: InvokeDirectiveRet	/reject(protocol error)				^PIF.InvokeDirectiveRet	
AIF: AsyncNotifyInv	/reject(protocol error)	^PIF.AsyncNotifyInv		/reject(unbind pend)	^PIF.AsyncNotifyInv	
PIF: ThrowEventInv	/reject(protocol error)	^AIF.ThrowEventInv ***{1}***		/reject(protocol error)	^AIF.ThrowEventInv ***{1}***	

	1 UNBOUND	2 READY			3 ACTIVE	
		2.1 BOUND	2.2 START PEND	2.3 UNBIND PEND	3.1 ACTIVE	3.2 STOP PEND
AIF: ThrowEventRet	/reject(protocol error)	^PIF.ThrowEventRet ***{1}***		/reject(unbind pend)	^PIF.ThrowEventRet ***{1}***	

NOTE – The operation THROW-EVENT is defined in the transfer services but the associated management support is not yet in place. As long as this situation exists, applications should respond with a return holding a negative result and the diagnostic ‘other reason’.

4.5.4 USER SIDE STATE TABLES

4.5.4.1 States

All user side state tables use the same set of states. The main states are identical to those defined in the CCSDS Recommended Standards for SLE transfer services. Sub-states have been added to allow presentation of further details related to the interactions with the application and the proxy. The states are defined as follows.

UNBOUND:UNBOUND	The user is not bound to the service instance.
UNBOUND:BIND PEND	A BIND invocation has been issued, the service provider has not yet responded.
READY:BOUND	A BIND return with a positive result has been received and no START invocation has been sent, or a STOP operation has been completed.
READY:START PEND	A START invocation has been issued, the service provider has not yet responded.
READY:UNBIND PEND	An UNBIND invocation has been issued, the service provider has not yet responded.
ACTIVE:ACTIVE	A START return with a positive result has been received.
ACTIVE:STOP PEND	A STOP invocation has been issued, the service provider has not yet responded.

4.5.4.2 Common State Table—User Initiated Binding

4.5.4.2.1 Events

4.5.4.2.1.1 Events received from the Application Interface (`ISLE_ServiceInitiate`)

BindInv	call to `InitiateOpInvoke()` with a BIND operation
UnbindInv	call to `InitiateOpInvoke()` with a UNBIND operation
GetPrmInv	call to `InitiateOpInvoke()` with a GET-PARAMETER operation
ScheduleStatRepInv	call to `InitiateOpInvoke()` with a SCHEDULE-STATUS-REPORT operation
PeerAbortInv	call to `InitiateOpInvoke()` with a PEER-ABORT operation

4.5.4.2.1.2 Events sent to the Application Interface (`ISLE_ServiceInform`)

BindRet	call to `InformOpReturn()` with a BIND operation
UnbindRet	call to `InformOpReturn()` with a UNBIND operation
GetPrmRet	call to `InformOpReturn()` with a GET-PARAMETER operation
ScheduleStatRepRet	call to `InformOpReturn()` with a SCHEDULE-STATUS-REPORT operation
StatusReportInv	call to `InformOpInvoke()` with a STATUS-REPORT operation
PeerAbortInv	call to `InformOpInvoke()` with a PEER-ABORT operation
ProtocolAbort	call to `ProtocolAbort()`

4.5.4.2.1.3 Events received from the Proxy Interface (`ISLE_SrvProxyInform`)

BindRet	call to `InformOpReturn()` with a BIND operation
UnbindRet	call to `InformOpReturn()` with a UNBIND operation
GetPrmRet	call to `InformOpReturn()` with a GET-PARAMETER operation
ScheduleStatRepRet	call to `InformOpReturn()` with a SCHEDULE-STATUS-REPORT operation
StatusReportInv	call to `InformOpInvoke()` with a STATUS-REPORT operation
PeerAbortInv	call to `InformOpInvoke()` with a PEER-ABORT operation
ProtocolAbort	call to `ProtocolAbort()`

4.5.4.2.1.4 Events sent to the Proxy Interface (`ISLE_SrvProxyInitiate`)

BindInv	call to `InitiateOpInvoke()` with a BIND operation
UnbindInv	call to `InitiateOpInvoke()` with a UNBIND operation
GetPrmInv	call to `InitiateOpInvoke()` with a GET-PARAMETER operation
ScheduleStatRepInv	call to `InitiateOpInvoke()` with a SCHEDULE-STATUS-REPORT operation
PeerAbortInv	call to `InitiateOpInvoke()` with a PEER-ABORT operation

4.5.4.2.1.5 Internal Events

Peer Abort	peer abort event generated by a pre-processing function or a post-processing function
Return timeout	the time to wait for a specific return-PDU has elapsed

4.5.4.2.2 Predicates

result = positive	The result parameter in the PDU indicates ‘positive result’.
result = negative	The result parameter in the PDU indicates ‘negative result’.

4.5.4.2.3 Actions

4.5.4.2.3.1 Discrete Actions

/reject(reason)	Reject the event by returning an error code to the function invoking the event.
/clear local returns	Clear the list of pending local returns and release operation objects.

4.5.4.2.3.2 Compound Actions

/ABORT

Abort processing is forward/return-service specific, see /ABORT in 4.5.4.3 and 4.5.4.4.

/CLEANUP

Cleanup processing is forward/return-service specific, see /CLEANUP in 4.5.4.3 and 4.5.4.4.

4.5.4.2.4 Common State Table—User Side

	1 UNBOUND		2 READY			3 ACTIVE
	1.1 UNBOUND	**1.2 BIND PEND**	**2.1 BOUND**	**2.2 START PEND**	**2.3 UNBIND PEND**	
AIF: BindInv	^PIF.BindInv → 1.2	/reject(protocol error)				
PIF: BindRet	/reject(protocol error)	^AIF.BindRet [result = positive] → 2.1 [result = negative] → 1.1	/reject(protocol error)			
AIF: UnbindInv	/reject(protocol error)		^PIF.UnbindInv /clear local returns → 2.3	/reject(protocol error)		
PIF: UnbindRet	/reject(protocol error)				^AIF.UnbindRet /CLEANUP → 1.1	/reject(protocol error)
AIF: GetPrmInv	/reject(protocol error)		^PIF.GetPrmInv		/reject(protocol err)	^PIF.GetPrmInv
PIF: GetPrmRet	/reject(protocol error)		^AIF.GetPrmRet			***{1}***
AIF: ScheduleStatRepInv	reject(protocol error)		^PIF.ScheduleStatRepInv		/reject(protocol err)	^PIF.ScheduleStatRepInv
PIF: ScheduleStatRepRet	/reject(protocol error)		^AIF.ScheduleStatRepRet			***{1}***
PIF: StatusReportInv	/reject(protocol error)		^AIF.StatusReportInv			***{2}***
PIF: PeerAbortInv	/reject(protocol error)	^AIF.PeerAbortInv /CLEANUP → 1.1				

	1 UNBOUND		2 READY			3 ACTIVE
	1.1 UNBOUND	**1.2 BIND PEND**	**2.1 BOUND**	**2.2 START PEND**	**2.3 UNBIND PEND**	
AIF: PeerAbortInv	/reject(protocol error)	^PIF.PeerAbortInv /CLEANUP → 1.1				
PIF: ProtocolAbort	/reject(protocol error)	^AIF.ProtoclAbortInv /CLEANUP → 1.1				
INT: Return timeout	N/A	/ABORT (return timeout) → 1.1				
INT: PeerAbort(reason)	N/A	/ABORT(reason) → 1.1				

NOTES

1 In the state UNBIND-PENDING, returns can still be received from the peer.

2 In the state UNBIND-PENDING, no further invocations should be sent by the peer. However, the peer may not yet have seen the UNBIND invocation. Therefore all invocations are passed to the application. The application should no longer respond.

4.5.4.3 Return Link SLE Services

4.5.4.3.1 Events

4.5.4.3.1.1 Events received from the Application Interface (`ISLE_ServiceInitiate`)

StartInv	call to `InitiateOpInvoke()` with a START operation
StopInv	call to `InitiateOpInvoke()` with a STOP operation

4.5.4.3.1.2 Events sent to the Application Interface (`ISLE_ServiceInform`)

StartRet	call to `InformOpReturn()` with a START operation
StopRet	call to `InformOpReturn()` with a STOP operation
TransferDataInv	call to `InformOpInvoke()` with a TRANSFER-DATA operation
SyncNotifyInv	call to `InformOpInvoke()` with a SYNC-NOTIFY operation
PeerAbortInv	call to `InformOpInvoke()` with a PEER-ABORT operation

4.5.4.3.1.3 Events received from the Proxy Interface (`ISLE_SrvProxyInform`)

StartRet	call to `InformOpReturn()` with a START operation
StopRet	call to `InformOpReturn()` with a STOP operation
TransferBufferInv	call to `InformOpInvoke()` with a TRANSFER-BUFFER operation.

4.5.4.3.1.4 Events sent to the Proxy Interface (`ISLE_SrvProxyInitiate`)

StartInv	call to `InitiateOpInvoke()` with a START operation
StopInv	call to `InitiateOpInvoke()` with a STOP operation
PeerAbortInv	call to `InitiateOpInvoke()` with a PEER-ABORT operation

4.5.4.3.2 Predicates

result = positive	The result parameter in the PDU indicates ‘positive result’.
result = negative	The result parameter in the PDU indicates ‘negative result’.
buffer empty	The transfer buffer is empty.
pdu = data	The PDU extracted from the transfer buffer is a TRANSFER-DATA invocation.
pdu = notification	The PDU extracted from the transfer buffer is a SYNC-NOTIFY invocation.

4.5.4.3.3 Actions

4.5.4.3.3.1 Discrete Actions

/reject(reason)	Reject the event by returning an error code to the function invoking the event.
/extract pdu	Extract (and remove) the PDU at the beginning of the transfer buffer.
/clear remote returns	Cancel return timers for all pending remote returns, clear the list of pending remote returns, and release operation objects.
/clear local returns	Clear the list of pending local returns and release operation objects.

4.5.4.3.3.2 Compound Actions

/ABORT(diagnostic) is defined as

```
^PIF.PeerAbort(diagnostic)
^AIF.PeerAbort(diagnostic)
/CLEANUP
```

/CLEANUP is defined as

```
/clear remote returns
/clear local returns
```

/PROCESS BUFFER is defined as

```
WHILE not buffer empty DO
   /extract pdu
   IF pdu = data THEN
      ^AIF.TransferDataInv
   ELSE
      IF pdu = notification THEN
         ^AIF.SyncNotifyInv
      ELSE
         /ABORT(protocol error)
      END IF
   END IF
END WHILE
```

4.5.4.3.4 Return Link State Table—User Side

	1 UNBOUND	2 READY			3 ACTIVE	
		2.1 BOUND	2.2 START PEND	2.3 UNBIND PEND	3.1 ACTIVE	3.2 STOP PEND
AIF: StartInv	/reject(protocol error)	^PIF.StartInv → 2.2	/reject(protocol error)			
PIF: StartRet	/reject(protocol error)	/ABORT(protocol error) → 1.1	^AIF.StartRet [result = positive] → 3.1 [result = negative] → 2.1	/ABORT(protocol error) → 1.1		
AIF: StopInv	/reject(protocol error)				^PIF.StopInv → 3.2	/reject(protocol error)
PIF: StopRet	/reject(protocol error)	/ABORT(protocol error) → 1.1				^AIF.StopRet [result = positive] → 2.1 [result = negative] → 3.1
PIF: TransferBufferInv	/reject(protocol error)	/ABORT(protocol error) → 1.1			/PROCESS BUFFER	

4.5.4.4 Forward Link SLE Services

4.5.4.4.1 Events

4.5.4.4.1.1 Events received from the Application Interface (`ISLE_ServiceInitiate`)

StartInv	call to `InitiateOpInvoke()` with a START operation
StopInv	call to `InitiateOpInvoke()` with a STOP operation
TransferDataInv	call to `InitiateOpInvoke()` with a TRANSFER-DATA operation
InvokeDirectiveInv	call to `InitiateOpInvoke()` with an INVOKE-DIRECTIVE operation
ThrowEventInv	call to `InitiateOpInvoke()` with a THROW-EVENT operation

4.5.4.4.1.2 Events sent to the Application Interface (`ISLE_ServiceInform`)

StartRet	call to `InformOpReturn()` with a START operation
StopRet	call to `InformOpReturn()` with a STOP operation
TransferDataRet	call to `InformOpReturn()` with a TRANSFER-DATA operation
InvokeDirectiveRet	call to `InformOpReturn()` with an INVOKE-DIRECTIVE operation
ResumeDataTransfer	call to `ResumeDataTransfer()`
AsyncNotifyInv	call to `InformOpInvoke()` with an ASYNC-NOTIFY operation
ThrowEventRet	call to `InformOpReturn()` with a THROW-EVENT operation
PeerAbortInv	call to `InformOpInvoke()` with a PEER-ABORT operation

4.5.4.4.1.3 Events received from the Proxy Interface (`ISLE_SrvProxyInform`)

StartRet	call to `InformOpReturn()` with a START operation
StopRet	call to `InformOpReturn()` with a STOP operation
TransferDataRet	call to `InformOpReturn()` with a TRANSFER-DATA operation
InvokeDirectiveRet	call to `InformOpReturn()` with an INVOKE-DIRECTIVE operation
AsyncNotifyInv	call to `InformOpInvoke()` with an ASYNC-NOTIFY operation
ThrowEventRet	call to `InformOpReturn()` with a THROW-EVENT operation
PDUTransmitted	call to `PDUTransmitted()`

4.5.4.4.1.4 Events sent to the Proxy Interface (`ISLE_SrvProxyInitiate`)

StartInv	call to `InitiateOpInvoke()` with a START operation
StopInv	call to `InitiateOpInvoke()` with a STOP operation
TransferDataInv	call to `InitiateOpInvoke()` with a TRANSFER-DATA operation
InvokeDirectiveInv	call to `InitiateOpInvoke()` with an INVOKE-DIRECTIVE operation
ThrowEventInv	call to `InitiateOpInvoke()` with a THROW-EVENT operation
PeerAbortInv	call to `InitiateOpInvoke()` with a PEER-ABORT operation

4.5.4.4.2 Predicates

result = positive	The result parameter in the PDU indicates ‘positive result’.
result = negative	The result parameter in the PDU indicates ‘negative result’.
data transmitted	The TRANSFER-DATA invocation passed to the proxy could be transmitted immediately as indicated by the return code of the function.
data queued	A TRANSFER-DATA invocation has been passed to the proxy for transfer and the PDU Transmitted event has not yet been received for that buffer.

4.5.4.4.3 Actions

4.5.4.4.3.1 Discrete Actions

/reject(reason)	Reject the event by returning an error code to the function invoking the event.
/clear remote returns	Cancel return timers for all pending remote returns, clear the list of pending remote returns, and release operation objects.
/clear local returns	Clear the list of pending local returns and release operation objects.
/suspend data transfer	Request the application to suspend data transfer, by returning the appropriate code from the function transmitting the event.

4.5.4.4.3.2 Compound Actions

/ABORT(diagnostic) is defined as

```
^PIF.PeerAbort(diagnostic)
^AIF.PeerAbort(diagnostic)
/CLEANUP
```

/CLEANUP is defined as

```
/clear remote returns
/clear local returns
/set data queued = FALSE
```

/PROCESS TD INV is defined as

```
IF data queued THEN
   /reject(transfer suspended)
ELSE
   ^PIF.TransferDataInv
   IF not data transmitted THEN
      /set data queued = TRUE
      /suspend data transfer
   END IF
END IF
```

NOTE – Processing as specified here, applies to a single outstanding TRANSFER DATA invocation. Multiple outstanding TRANSFER DATA invocations might be used by an implementation to increase performance.

/PROCESS PDU TRANSMITTED is defined as

```
IF data queued THEN
   /set data queued = FALSE
   ^AIF.ResumeDataTransfer
END IF
```

4.5.4.4.4 Forward Link State Table—User Side

	1 UNBOUND	2 READY			3 ACTIVE	
		2.1 BOUND	2.2 START PEND	2.3 UNBIND PEND	3.1 ACTIVE	3.2 STOP PEND
AIF: StartInv	/reject(protocol error)	^PIF.StartInv → 2.2	/reject(protocol error)			
PIF: StartRet	/reject(protocol error)	/ABORT (protocol error) → 1.1	^AIF.StartRet [result = positive] → 3.1 [result = negative] → 2.1	/ABORT(protocol error) → 1.1		
AIF: StopInv	/reject(protocol error)				^PIF.StopInv → 3.2	/reject(protocol error)
PIF: StopRet	/reject(protocol error)	/ABORT(protocol error) → 1.1				^AIF.StopRet [result = positive] → 2.1 [result = negative] → 3.1
AIF: TransferDataInv	/reject(protocol error)				/PROCESS TD INV	/reject(protocol error)
PIF: TransferDataRet	/reject(protocol error)	/ABORT(protocol error) → 1.1			^AIF.TransferDataRet	
AIF: InvokeDirectiveInv	/reject(protocol error)				^PIF.InvokeDirectiveInv	/reject(protocol error)
PIF: InvokeDirectiveRet	/reject(protocol error)	/ABORT(protocol error) → 1.1			^AIF.InvokeDirectiveRet	
PIF: PDU transmitted	/reject				/PROCESS PDU TRANSMITTED	
PIF: AsyncNotifyInv	/reject(protocol error)	^AIF.AsyncNotifyInv				***{2}***

	1 UNBOUND	2 READY			3 ACTIVE	
		2.1 BOUND	2.2 START PEND	2.3 UNBIND PEND	3.1 ACTIVE	3.2 STOP PEND
AIF: ThrowEventInv	/reject(protocol error)	^PIF.ThrowEventInv		/reject(protocol err)	^PIF.ThrowEventInv	
PIF: ThrowEventRet	/reject(protocol error)	^AIF.ThrowEventRet				***{1}***

NOTES

1 In the state UNBIND-PENDING, returns can still be received from the peer.

2 In the state UNBIND-PENDING, no further invocations should be sent by the peer. However, the peer may not yet have seen the UNBIND invocation. Therefore all invocations are passed to the application. The application should no longer respond.

ANNEX A

SPECIFICATION OF COMMON INTERFACES

(Normative)

A1 INTRODUCTION

This annex contains the C++ definition of interfaces that are common for all SLE service types and of supporting types required by these interfaces. Service-type specific interfaces are defined by the relevant supplemental Recommended Practice documents for service-specific APIs.

The interface specifications are structured according to the components that must provide the implementation:

a) Interfaces implemented by the component 'SLE Utilities' are defined in subsection A4.

b) Interfaces implemented by the component 'SLE Operations' are defined in subsection A5.

c) Interfaces implemented by the component 'API Proxy' are defined in subsection A7.

d) Interfaces implemented by the component 'API Service Element' are defined in subsection A8.

e) Interfaces that must be provided by the SLE Application are defined in subsection A9.

Interfaces that must be implemented by more than one component are defined in subsection A6. Interfaces defined in that subsection must be implemented by the component 'API Proxy' and 'API Service Element'.

Subsection A3 defines types used throughout the remaining subsections.

The conventions used for the specification are explained in subsection A2.

The specifications of this annex are complemented by the definition of the 'Simple Component Model' in annex D.

A2 CONVENTIONS

A2.1 OVERVIEW

The specification of the interfaces follows the design patterns and conventions described for the 'Simple Component Model' in annex D. In order to be consistent with those elements adopted from COM, the coding style has been also adopted from COM to a large extent.

A2.2 INTERFACES

The 'keyword' `interface` is defined by

```
#define interface struct
```

in the file `SLE_SCM.H` described in annex D. All interfaces are directly or indirectly derived from the interface `IUnknown,` which is also defined in `SLE_SCM.H`.

A2.3 NAMING CONVENTIONS

Names for the following items start with uppercase letters:

a) All types, i.e.:

   1) Interfaces (e.g., `ISLE_Bind`);

   2) enumeration types (e.g., `SLE_ParameterName`);

   3) other types declared by `typedef` (e.g., `SLE_InvokeId`);

b) Method names (e.g., `InitiateOpInvoke()`).

Names for the following items start with lowercase letters:

a) variables;

b) arguments of methods;

c) enumeration labels.

All names at global scope in this specification use the prefix '`SLE`' (or '`sle`' when the named item is supposed to start with a lowercase letter).

All interfaces start with a capital '`I`', such that interface names are prefixed with '`ISLE`'.

NOTE – The interface `IMalloc` defined in A4.3 is the only exception to this rule because of the considerations presented in A2.6.

Because enumeration labels are defined at global scope, the prefix is extended to include an abbreviation for the enumeration type. For instance, all labels of the enumeration type `SLE_ParameterName` are prefixed by '`slePN`'.

An underscore character is used to separate the prefix from the name. Within the name itself upper and lower case is used to improve readability.

A2.4 ACCESS TO OBJECT ATTRIBUTES

Methods that only provide read access to an attribute of an object, are named using the prefix '`Get`'. Methods that set the attribute are named using the prefix '`Set`'. An underscore character is used to separate the prefix from the name. For example the method to read the service type is called `Get_ServiceType()` and the method to set the service type is called `Set_ServiceType()`.

When the attribute type is not a basic type (e.g., a character string) the following conventions are applied:

a) If it can be assumed that the implementation stores the attribute in the format in which it is delivered, the return value is defined to be `const`. In these cases, the client must copy the value if it wants to modify it.

b) If the object implementation might have to derive the value a pointer to a not constant object is returned. In these cases the client must delete the returned value.

c) In order to optimize performance, an additional retrieval method, prefixed by '`Remove_`' is defined for attributes that might become large (e.g., the space link data). These methods return a pointer to the internal representation and remove that pointer from the object itself. The client calling that method must make sure the memory is released when the data are no longer needed.

d) For setting of attributes a pointer or reference to a `const` object is generally used. For potentially larger arguments, an additional method (prefixed by '`Put`') is defined which passes a pointer to a not constant object. In these cases the implementation is expected to delete the data passed with the argument, when it no longer needs it.

A2.5 CONDITIONAL AND OPTIONAL ATTRIBUTES

Attributes of operation objects can be:

a) conditional, i.e., their value is only defined when another attribute has a certain value;

b) optional, i.e., the value may or may not be defined.

For access to conditional attributes, checking of the condition is considered a precondition; i.e., the result of calling the access method is undefined when the attribute is not present.

For access to optional attributes, different approaches are used, depending on the type of the attribute:

a) For enumeration types, an additional enumeration value ‘undefined’ is added to the type declaration. This value is returned if the attribute is currently undefined.

b) Object types or composite C++ types, such as arrays or structures are returned via a pointer to a constant object, instead of using a reference to a constant object. If an optional attribute is undefined, the access method returns a NULL pointer.

c) For simple types the following cases are distinguished:

   1) if the valid range of attribute values does not include the complete range covered by the type, a special value is selected to indicate that the attribute is not defined;

   2) in other cases, a special method is provided to check whether the attribute is defined or not.

In cases where presence or absence of an attribute is identified by a special method, absence of an operation object attribute shall be marked as ‘(not used)’ in the tables specifying the initial values of operation object attributes.

A2.6 MEMORY MANAGEMENT

Non-object data structures, to which pointers are passed across component boundaries, might be created by one component or the application and released by another component or the application. Use of a consistent memory management scheme by all involved parties is of prime importance to ensure integrity of process memory.

Therefore, this Recommended Practice defines a specific memory management interface `IMalloc`, which must be used by all API components and by the application when creating or deleting data structures to which pointers are passed across component boundaries. The interface `IMalloc` is implemented by the component SLE Utilities. A pointer to the interface can be obtained using the method `CreateMemoryManager()` of the Utility Factory.

NOTES

1 Memory management for objects created by API components is controlled by the reference counting scheme for interfaces described in annex D. This scheme implies that the memory for such objects is always allocated and released by the same component. Therefore, the means by which memory is allocated and released for such objects is considered a local implementation issue and not prescribed by this Recommended Practice. The same applies to interfaces, which are implemented by objects within the application software.

2 Data structures, which might be created by one component and deleted by another component, are generally strings, arrays or structures passed by `Put_xxx()` or `Remove_xxx()` methods.

3 For data that are never passed across component boundaries and for data that are always passed by value, memory management is considered a local implementation issue and not prescribed by this Recommended Practice.

4 As specified in A2.4, this Recommended Practice applies the convention that data, which are passed across component boundaries using a reference to a constant data structure or a pointer to a constant data structure, must not be deleted by the calling software. Therefore, use of the memory manager interface `IMalloc` is not mandated if a data structure is only passed across component boundaries in these ways.

5 The specification of the interface `IMalloc` defines a subset of the COM interface `IMalloc`, in order to enable use of the SLE API in a COM environment. Further details concerning the use of this interface and the implementation of the interface in other environments can be found in A4.3 and annex D.

A2.7 INTERFACE IDENTIFIERS

Interface identifiers are displayed in the format as defined for the COM registry. In addition, each interface contains a macro that allows pre-setting of the structure GUID (see annex D).

The name of the macro is constructed as `IID_<interface-name>_DEF` Guidelines for use of this macro can be found in annex D.

A2.8 TYPE DEFINITIONS

Types other than interfaces are defined at global scope. They are grouped into two files, namely:

a) `SLE_Types.h` for types derived from the CCSDS Recommended Standards for SLE transfer services;

b) `SLE_APITypes.h` for types specified by the API.

All types are defined in a manner that is compatible with the C language in order to simplify mapping of the interfaces to C. For enumeration types derived from the CCSDS Recommended Standards for SLE transfer services the numbers assigned to the labels correspond to the integer values used in those specifications.

A2.9 RESULT CODES

This specification adopts the scheme to define result codes from COM. All values that can be used for the variable `HRESULT` are defined in annex B and in the file `SLE_Result.h`.

A2.10 FUNCTION OVERLOADING

In order to simplify mapping to the C language, this specification does not use overloaded functions except for overloaded operators. Overloaded operators can be mapped to function names in C.

A2.11 OBJECT CREATION METHODS

The signature of methods creating and returning objects follows the COM conventions:

a) the GUID of the interface is passed as an input argument (and checked by the implementation);

b) a pointer to the interface of the object is passed as an output argument of the type `void**;`

c) the method returns a result code.

A2.12 FILES

This specification defines header files that contain interface declarations and type definitions. Obviously, these definitions are not mandatory, but present a recommendation. A set of the files defined in this specification is available from the same source as the specification itself.

A3 TYPE DEFINITIONS

A3.1 SLE TYPES

A3.1.1 General

File `SLE_Types.h`

The following basic types have been derived from the ASN.1 modules in the CCSDS Recommended Standards for SLE transfer services. The source ASN.1 type is indicated in brackets. For all enumeration types a special value ‘invalid’ is defined, which is returned if the associated value in the operation object has not yet been set, or is not applicable in case of a choice.

The type definitions in this specification cover all those types that are common for all service types or for a subset of service types. These types are defined in the ASN.1 modules:

a) CCSDS-SLE-TRANSFER-SERVICE-COMMON-TYPES;

b) CCSDS-SLE-TRANSFER-SERVICE-BIND-TYPES; and

c) CCSDS-SLE-TRANSFER-COMMON-PDUS.

The definition of the SLE service parameters (`ParameterName`) has been excluded because extensions are expected when further SLE services are added. Service parameters are defined in the relevant supplemental Recommended Practice documents for service-specific APIs.

A3.1.2 Auxiliary Types

Size of Data

```
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif
```

On POSIX systems, `size_t` is defined by standard header files; redefinition must be prevented by conditional compilation.

Definition of an Octet

```
typedef unsigned char SLE_Octet;
```

In order to distinguish between character strings and sequences of octets (bytes) frequently used by SLE Service specifications, the API defines a special type for an octet. The type `char*` always refers to a zero terminated string of characters.

Yes/No Value

```
typedef enum SLE_YesNo
{
  sleYN_No       =  0,
  sleYN_Yes      =  1,
  sleYN_invalid  = -1
} SLE_YesNo;
```

The type describes a Boolean value, which might not be available at certain times.

A3.1.3 Types derived from CCSDS-SLE-TRANSFER-SERVICE-COMMON-TYPES

Delivery Mode [`DeliveryMode`]

```
typedef enum SLE_DeliveryMode
{
  sleDM_rtnTimelyOnline       =  0,
  sleDM_rtnCompleteOnline     =  1,
  sleDM_rtnOffline            =  2,
  sleDM_fwdOnline             =  3,
  sleDM_fwdOffline            =  4,
  sleDM_invalid               = -1
```

```
} SLE_DeliveryMode;
```

Common Diagnostics [`Diagnostics`]

```
typedef enum SLE_Diagnostics
{
  sleD_duplicateInvokeId       = 100,
  sleD_otherReason             = 127,
  sleD_invalid                 = -1
} SLE_Diagnostics;
```

Forward Data Unit Status [`ForwardDuStatus`]

```
typedef enum SLE_ForwardDuStatus
{
  sleFDS_radiated                   =   0,
  sleFDS_expired                    =   1,
  sleFDS_interrupted                =   2,
  sleFDS_acknowledged               =   3,
  sleFDS_productionStarted          =   4,
  sleFDS_productionNotStarted       =   5,
  sleFDS_unsupportedTransmissionMode =  6,
  sleFDS_invalid                    =  -1
} SLE_ForwardDuStatus;
```

Invocation Identifier [`InvokeId`]

```
typedef unsigned short SLE_InvokeId;
```

Generation of Notifications [`SlduStatusNotification`]

```
typedef enum SLE_SlduStatusNotification
{
   sleSN_produceNotification       =   0,
   sleSN_doNotProduceNotification  =   1,
   sleSN_invalid                   =  -1
} SLE_SlduStatusNotification;
```

A3.1.4 Types derived from CCSDS-SLE-TRANSFER-SERVICE-BIND-TYPES

Version Number [`VersionNumber`]

```
typedef unsigned short SLE_VersionNumber;
```

Duration [`Duration`]

```
typedef unsigned long SLE_Duration; /* in microseconds */
```

SLE Service Type [`ApplicationIdentifier`]

```
typedef enum SLE_ApplicationIdentifier
{
  sleAI_rtnAllFrames            =   0,
```

```
  sleAI_rtnInsert               =   1,
  sleAI_rtnChFrames             =   2,
  sleAI_rtnChFsh                =   3,
  sleAI_rtnChOcf                =   4,
  sleAI_rtnBitstr               =   5,
  sleAI_rtnSpacePkt             =   6,
  sleAI_fwdAosSpacePkt          =   7,
  sleAI_fwdAosVca               =   8,
  sleAI_fwdBitstr               =   9,
  sleAI_fwdProtoVcdu            =  10,
  sleAI_fwdInsert               =  11,
  sleAI_fwdVcdu                 =  12,
  sleAI_fwdTcSpacePkt           =  13,
  sleAI_fwdTcVca                =  14,
  sleAI_fwdTcFrame              =  15,
  sleAI_fwdCltu                 =  16,
  sleAI_invalid                 =  -1
} SLE_ApplicationIdentifier;
```

BIND Diagnostic [`BindDiagnostic`]

```
typedef enum SLE_BindDiagnostic
{
  sleBD_accessDenied                  =   0,
  sleBD_serviceTypeNotSupported       =   1,
  sleBD_versionNotSupported           =   2,
  sleBD_noSuchServiceInstance         =   3,
  sleBD_alreadyBound                  =   4,
  sleBD_siNotAccessibleToThisInitiator =   5,
  sleBD_inconsistentServiceType       =   6,
  sleBD_invalidTime                   =   7,
  sleBD_outOfService                  =   8,
  sleBD_otherReason                   = 127,
  sleBD_invalid                       =  -1
} SLE_BindDiagnostic;
```

UNBIND Reason [`UnbindReason`]

```
typedef enum SLE_UnbindReason
{
  sleUBR_end                          =   0,
  sleUBR_suspend                      =   1,
  sleUBR_versionNotSupported          =   2,
  sleUBR_otherReason                  = 127,
  sleUBR_invalid                      =  -1
} SLE_UnbindReason;
```

PEER-ABORT Diagnostic [`PeerAbortDiagnostic`]

```
typedef enum SLE_PeerAbortDiagnostic
{
  slePAD_accessDenied                 =   0,
  slePAD_unexpectedResponderId        =   1,
  slePAD_operationalRequirement       =   2,
  slePAD_protoclError                 =   3,
```

```
  slePAD_communicationsFailure         =   4,
  slePAD_encodingError                 =   5,
  slePAD_returnTimeout                 =   6,
  slePAD_endOfServiceProvisionPeriod   =   7,
  slePAD_unsolicitedInvokeId           =   8,
  slePAD_otherReason                   = 127,
  slePAD_invalid                       =  -1
} SLE_PeerAbortDiagnostic;
```

A3.1.5 Types derived from CCSDS-SLE-TRANSFER-COMMON-PDUS

Reporting Cycle [`ReportingCycle`]

```
typedef unsigned int SLE_ReportingCycle;
```

Report Request Type [`ReportRequestType`]

```
typedef enum SLE_ReportRequestType
{
  sleRRT_immediately               =   0,
  sleRRT_periodically              =   1,
  sleRRT_stop                      =   2,
  sleRRT_invalid                   =  -1
} SLE_ReportRequestType;
```

SCHEDULE-STATUS-REPORT Diagnostic [`DiagnosticScheduleStatusReport`]

```
typedef enum SLE_ScheduleStatusReportDiagnostic
{
  sleSSD_notSupportedInThisDeliveryMode =   0,
  sleSSD_alreadyStopped                 =   1,
  sleSSD_invalidReportingCycle          =   2,
  sleSSD_invalid                        =  -1
} SLE_ScheduleStatusReportDiagnostic;
```

A3.2 SLE API TYPES

File `SLE_APITypes.h`

The following types are used throughout the API Specification.

State of an Association

```
typedef enum SLE_AssocState
{
  sleAST_unbound            = 0,
  sleAST_bindPending        = 1,   /* Bind initiated remotely   */
  sleAST_bound              = 2,
  sleAST_remoteUnbindPending = 3,   /* Unbind initiated remotely */
  sleAST_localUnbindPending  = 4    /* Unbind initiated locally  */
} SLE_AssocState;
```

State of a Service Instance

```
typedef enum SLE_SIState
{
  sleSIS_unbound            = 0,
  sleSIS_bindPending        = 1,
  sleSIS_bound              = 2,
  sleSIS_unbindPending      = 3,
  sleSIS_startPending       = 4,
  sleSIS_active             = 5,
  sleSIS_stopPending        = 6
} SLE_SIState;
```

Role of an SLE Application

```
typedef enum SLE_AppRole
{
  sleAR_user               = 0,
  sleAR_provider           = 1,
  sleAR_userAndProvider    = 2
} SLE_AppRole;
```

Role of an SLE Application in the BIND Operation

```
typedef enum SLE_BindRole
{
  sleBR_initiator              = 0,
  sleBR_responder              = 1,
  sleBR_initiatorAndResponder  = 2
} SLE_BindRole;
```

Port Registration Identifier

```
typedef void* SLE_PortRegId;
```

API Components

```
typedef enum SLE_Component
{
  sleCP_application            = 0,
  sleCP_serviceElement         = 1,
  sleCP_proxy                  = 2,
  sleCP_operations             = 3,
  sleCP_utilities              = 4

} SLE_Component;
```

Authentication Mode

```
typedef enum SLE_AuthenticationMode
{
  sleAM_none        = 0, /* authentication not used           */
  sleAM_bindOnly    = 1, /* authetication only for bind       */
  sleAM_all         = 2  /* authentication for all operations */
} SLE_AuthenticationMode;
```

Event Handle

```
#if defined(_SLE_EH_FILE_DESCRIPTOR_)   /* UNIX file descriptor */
typedef enum SLE_EventType
{
  sleET_readEvent    = 0;
  sleET_writeEvent   = 1;
  sleET_exception    = 2;
} SLE_EventType;

typedef struct SLE_EventHandle
{
  int filedes;                          /* file descriptor */
  SLE_EventType eventType;
} SLE_EventHandle;

#elif defined(_SLE_EH_EVENT_FLAG_)      /* VMS event flag */
typedef unsigned int SLE_EventHandle;

#elif defined (_SLE_EH_EVENT_OBJECT_)   /* Win32 event object */
typedef HANDLE SLE_EventHandle;

/* further definitions may be added in future versions */
#else
typedef void* SLE_EventHandle;
#endif
```

The format of the event handle depends on the processing platform and the operating system features selected by the implementation of the API component:

a) On UNIX, the event handle consists of a file descriptor, and the type of events supported by the `select()` call. Note that two event handles are considered to refer to the same event specification only when the file descriptor and the event type match. Different event types on the same file descriptor are considered unrelated. (This version can also be used on other platforms supporting `select()` in the combination with the socket API.)

b) On Windows systems, event objects can be used.

Timer Identifier

```
typedef void* SLE_TimerId;
```

The identifier for a timer supported by the interface `ISLE_TimerHandler`.

Trace Level

```
typedef enum SLE_TraceLevel
{
  sleTL_low      = 0,  /* only state changes                */
  sleTL_medium   = 1,  /* plus all PDUs and internal events */
  sleTL_high     = 2,  /* plus arguments of the PDU         */
  sleTL_full     = 3   /* plus encoded data                 */
} SLE_TraceLevel;
```

Log Message Types

```
typedef enum SLE_LogMessageType
{
  sleLM_alarm          = 0,
  sleLM_information    = 1
} SLE_LogMessageType;
```

Alarms Notified by the API

```
typedef enum SLE_Alarm
{
  sleAL_accessViolation  = 0,
  sleAL_authFailure      = 1,
  sleAL_commsFailure     = 2,
  sleAL_localAbort       = 3,
  sleAL_remoteAbort      = 4
} SLE_Alarm;
```

Variants of the CCSDS ASCII Time Format

```
typedef enum SLE_TimeFmt
{
  sleTF_dayOfMonth        = 0,
  sleTF_dayOfYear         = 1
} SLE_TimeFmt;
```

Time Resolution

```
typedef enum SLE_TimeRes
{
  sleTR_minutes           = 0,
  sleTR_seconds           = 1,
  sleTR_hundredMilliSec   = 2,
  sleTR_tenMilliSec       = 3,
  sleTR_milliSec          = 4,
  sleTR_hundredMicroSec   = 5,
  sleTR_tenMicroSec       = 6,
  sleTR_microSec          = 7
} SLE_TimeRes;
```

Operation Type

```
typedef enum SLE_OpType
{
  sleOT_bind                  =  0,
  sleOT_unbind                =  1,
  sleOT_peerAbort             =  2,
  sleOT_start                 =  3,
  sleOT_stop                  =  4,
  sleOT_transferData          =  5,
  sleOT_transferBuffer        =  6,
  sleOT_syncNotify            =  7,
  sleOT_asyncNotify           =  8,
  sleOT_scheduleStatusReport  =  9,
  sleOT_statusReport          = 10,
```

```
  sleOT_getParameter           = 11,
  sleOT_throwEvent             = 12,
  sleOT_invokeDirective        = 13
} SLE_OpType;
```

This enumeration specifies all operation types used by SLE Services. Not all operations are valid for all service types. In addition, operations for different service types differ. Therefore, an operation object is fully specified only by the combination of ‘operation type’ and ‘service type’.

Operation Result

```
typedef enum SLE_Result
{
  sleRES_positive       =  0,
  sleRES_negative       =  1,
  sleRES_invalid        = -1
} SLE_Result;
```

The result currently stored in a confirmed operation object.

Diagnostic Type

```
typedef enum SLE_DiagnosticType
{
  sleDT_noDiagnostics         = 0,
  sleDT_commonDiagnostics     = 1,
  sleDT_specificDiagnostics   = 2
} SLE_DiagnosticType;
```

The type of diagnostic stored in a confirmed operation object.

Originator of a Peer Abort

```
typedef enum SLE_AbortOriginator
{
  sleAO_peer            =  0, /* the peer system           */
  sleAO_proxy           =  1, /* the local proxy           */
  sleAO_serviceElement  =  2, /* the local service element */
  sleAO_application     =  3, /* the local application     */
  sleAO_invalid         = -1
} SLE_AbortOriginator;
```

A4 SLE UTILITY CLASSES

A4.1 COMPONENT CREATOR FUNCTION

File `<impl-id>.H`

The component implementing SLE utility classes includes a function to obtain a pointer to the utility object factory interface. The signature of this function is defined as:

```
extern "C" HRESULT
  <impl-id>_CreateUtilFactory( const GUID& iid,
                               ISLE_TimeSource* ptimeSource,
                               void** ppv );
```

where `<impl-id>` is replaced by the product identifier of the implementation. Note that external 'C' linkage is required. The function checks the argument identifying the factory interface and returns an error when the implementation does not support that identifier.

If a pointer to the interface `ISLE_TimeSource` is supplied, the component uses this interface to obtain the current time via the interface `ISLE_Time`. If a NULL pointer is supplied, the component uses system time.

Arguments

`iid`	identifier of the required interface
`ptimeSource`	pointer to the interface `ISLE_TimeSource`
`ppv`	pointer to the requested interface of the Utility Factory

Result codes

`S_OK`	the object has been created
`E_NOINTERFACE`	the specified interface is not supported

A4.2 SLE UTILITY FACTORY

Name `ISLE_UtilFactory`
GUID `{DED624E1-54CB-11d8-9CF5-0004761E8CFB}`
Inheritance: `IUnknown`
File `ISLE_UtilFactory.H`

The Utility Factory provides the means to create an SLE Utility object with a default initialization. The factory uses the interface identifier to verify that it can create the requested version of the object. If the IID is unknown, the factory returns an error. The lifetime of utility objects is controlled by reference counting as defined in annex D.

Synopsis

```
#include <SLE_SCM.H>
```

```
interface IMalloc;
interface ISLE_Time;
interface ISLE_SII;
interface ISLE_Credentials;
interface ISLE_SecAttributes;

#define IID_ISLE_UtilFactory_DEF { 0xded624e1, 0x54cb, 0x11d8, \
          { 0x9c, 0xf5, 0x0, 0x4, 0x76, 0x1e, 0x8c, 0xfb } };

interface ISLE_UtilFactory : IUnknown
{
  virtual HRESULT
    CreateMemoryManager( const GUID& iid,
                         void** ppv ) const = 0;
  virtual HRESULT
    CreateTime( const GUID& iid,
                void** ppv ) const = 0;
  virtual HRESULT
    CreateSII( const GUID& iid,
               void** ppv ) const = 0;
  virtual HRESULT
    CreateCredentials( const GUID& iid,
                       void** ppv ) const = 0;
  virtual HRESULT
    CreateSecAttributes( const GUID& iid,
                         void** ppv ) const = 0;
};
```

Methods

`HRESULT CreateMemoryManager( const GUID& iid, void** ppv ) const;`

Creates a new memory manager object which implements the COM interface `IMalloc`.

Arguments

`iid`	identifier of the required interface
`ppv`	pointer to the requested interface of the object

Result codes

`S_OK`	the object has been created
`E_NOINTERFACE`	the specified interface is not supported

`HRESULT CreateTime( const GUID& iid, void** ppv ) const;`

Creates a new time object, set to current time. Current time is obtained from the interface `ISLE_TimeSource`, if this interface was supplied to the creator-function of the component. Otherwise, the component uses system time.

Arguments

`iid`	identifier of the required interface
`ppv`	pointer to the requested interface of the object

Result codes

S_OK	the object has been created
E_NOINTERFACE	the specified interface is not supported

HRESULT CreateSII(const GUID& iid, void ppv) const;**

Creates a new service instance identifier object.

Arguments

iid	identifier of the required interface
ppv	pointer to the requested interface of the object

Result codes

S_OK	the object has been created
E_NOINTERFACE	the specified interface is not supported

HRESULT CreateCredentials(const GUID& iid, void ppv) const;**

Creates a new credentials object.

Arguments

iid	identifier of the required interface
ppv	pointer to the requested interface of the object

Result codes

S_OK	the object has been created
E_NOINTERFACE	the specified interface is not supported

HRESULT CreateSecAttributes(const GUID& iid, void ppv) const;**

Creates a new object holding security attributes.

Arguments

iid	identifier of the required interface
ppv	pointer to the requested interface of the object

Result codes

S_OK	the object has been created
E_NOINTERFACE	the specified interface is not supported

A4.3 SLE MEMORY MANAGER

Name `IMalloc`
GUID `{00000002-0000-0000-C000-000000000046}`
Inheritance: `IUnknown`
File `IMalloc.H`

The Memory Manager manages dynamic allocation and release of memory blocks. It must be used for all data structures passed over components boundaries or between the SLE API and the SLE application.

This interface conforms to the definition in the COM specification (including the GUID) in order to allow use of the COM memory manager in a COM environment. However, in the context of the SLE API, only the methods `Alloc()`, `Realloc()` and `Free()` are needed and implementations may provide dummy implementations of the methods `GetSize()`, `HeapMinimize()`, and `DidAlloc()`. Clients must not rely on these methods.

A more detailed discussion of memory management is provided in annex D.

Synopsis

```
#include <SLE_SCM.H>

#define IID_IMalloc_DEF { 0x00000002, 0x0000, 0x0000, \
          { 0xc0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x46 } }

interface IMalloc : IUnknown
{
  virtual void *
    Alloc( unsigned long cb ) = 0;
  virtual void *
    Realloc( void* pv, unsigned long cb ) = 0;
  virtual void
    Free( void* pv ) = 0;
  virtual unsigned long
    GetSize( void* pv ) = 0;
  virtual int
    DidAlloc( void* pv ) = 0;
  virtual void
    HeapMinimize() = 0;
};
```

Methods

`virtual void * Alloc( unsigned long cb );`

Allocates a memory block of at least `cb` bytes. The initial content of the returned memory block is undefined. Specifically, it is not guaranteed that the block is zeroed. The block actually allocated may be larger than `cb` bytes because of space required for alignment and for maintenance information.

Arguments

`cb`	minimum size (in bytes) of the memory block to be allocated; if `cb` is 0, `Alloc()` allocates a zero-length item and returns a valid pointer to that item

Results

`NULL`	insufficient memory available
`not NULL`	start address of the allocated memory block

`virtual void * Realloc( void* pv, unsigned long cb );`

Changes the size of a previously allocated memory block. The content of the block is unchanged up to the shorter of the new and old sizes, although the new block may be in a different location. Because the new block can be in a new memory location, the pointer returned by `Realloc()` is not guaranteed to be the pointer passed through the `pv` argument. If `pv` is not NULL and `cb` is 0, then the memory pointed to by `pv` is freed.

`Realloc()` returns a void pointer to the reallocated (and possibly moved) memory block. The return value is NULL if the size is zero and the buffer argument is not NULL, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, a type cast on the return value must be used.

Arguments

`pv`	current start address of the memory block to be reallocated; if pv is `NULL`, `Realloc()` functions in the same way as `Alloc()` and allocates a new block of `cb` bytes; if `pv` is not `NULL`, it should be a pointer returned by a prior call to `Alloc()`
`cb`	minimum new size (in bytes) of reallocated memory block

Results

`NULL`	insufficient memory available, or original memory block has been freed (if `cb` was 0)
`not NULL`	start address of reallocated memory block

`virtual void Free( void* pv );`

Deallocates a memory block. The `pv` argument points to a memory block previously allocated through a call to `Alloc()` or `Realloc()`. The number of bytes freed is the number of bytes with which the block was originally allocated (or reallocated, in the case of

`Realloc()`). After the call, the `pv` parameter is invalid, and can no longer be used. `pv` may be NULL, in which case this method does nothing.

Arguments

`pv`	address of memory block to be deallocated

`virtual unsigned long GetSize( void* pv );`

Returns the size (in bytes) of a memory block previously allocated with `IMalloc::Alloc()` or `IMalloc::Realloc()`.

NOTE – Implementations of the SLE Utilities component are not required to support this feature and may provide a dummy implementation, which always returns zero.

Arguments

`pv`	address of the memory block for which the size should be returned

`virtual int DidAlloc( void* pv );`

Determines if this allocator was used to allocate the specified block of memory.

NOTE – Implementations of the SLE Utilities component are not required to support this feature and may provide a dummy implementation, which always returns –1.

Arguments

`pv`	address of the memory block for which the query is made

Results

`1`	The memory block was allocated by this `IMalloc` instance
`0`	The memory block was not allocated by this `IMalloc` instance
`-1`	`DidAlloc()` is unable to determine whether or not it allocated the memory block

`virtual void HeapMinimize();`

Minimizes the heap as much as possible by releasing unused memory to the operating system, coalescing adjacent free blocks and committing free pages.

NOTE – Implementations of the SLE Utilities component are not required to support this feature and may provide a dummy implementation, which does nothing.

A4.4 SLE TIME

Name ISLE_Time
GUID {73517E80-D3F3-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_Time.H

Objects exporting this interface store time information with a resolution of up to one microsecond. They support input and output in the following formats:

a) CCSDS day segmented time code (CDS);

b) CCSDS ASCII Calendar Segmented Time Code.

They provide methods for comparison of times and calculation of the difference between two times measured in seconds and fractions of seconds.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

#define IID_ISLE_Time_DEF { 0x73517e80, 0xd3f3, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } };

interface ISLE_Time : IUnknown
{
  virtual HRESULT
    Set_CDS( const SLE_Octet* time ) = 0;
  virtual SLE_Octet*
    Get_CDS() const = 0;
  virtual HRESULT
    Set_DateAndTime( const char* dateAndTime ) = 0;
  virtual HRESULT
    Set_Time( const char* time ) = 0;
  virtual char*
    Get_Date( SLE_TimeFmt fmt ) const = 0;
  virtual char*
    Get_Time( SLE_TimeFmt fmt,
              SLE_TimeRes res = sleTR_seconds ) const = 0;
  virtual char*
    Get_DateAndTime( SLE_TimeFmt fmt,
                     SLE_TimeRes res = sleTR_seconds ) const = 0;
  virtual void
    Update() = 0;
  virtual double
    operator- ( const ISLE_Time& time ) const = 0;
  virtual bool
    operator== ( const ISLE_Time& time ) const = 0;
  virtual bool
    operator!= ( const ISLE_Time& time ) const = 0;
  virtual bool
    operator< ( const ISLE_Time& time ) const = 0;
  virtual bool
```

```
    operator> ( const ISLE_Time& time ) const = 0;
  virtual bool
    operator<= ( const ISLE_Time& time ) const = 0;
  virtual bool
    operator>= ( const ISLE_Time& time ) const = 0;
  virtual ISLE_Time*
    Copy() const = 0;
};
```

Methods

`HRESULT Set_CDS( const SLE_Octet* time );`

Sets the time to the value of the argument presented in the CCSDS day segmented time code.

Arguments

`time`	time coded according to the CCSDS day segmented time code, consisting of 8 octets; the P-Field is implicit and not included

Result codes

`S_OK`	the time has been set
`E_INVALIDARG`	the argument does not contain the expected format
`E_FAIL`	failure to set the time because of other reasons

`SLE_Octet* Get_CDS() const;`

Returns the time in the CCSDS day segmented time code, consisting of 8 octets; the P-Field is implicit and not included. The returned value must be deleted by the client.

`HRESULT Set_DateAndTime( const char* dateAndTime );`

Sets the date and time to the value specified by the input argument. The ASCII string can be coded in either variant A or B of the CCSDS ASCII Calendar Segmented Time Code. The time subset must contain at least the hours. The trailing ‘Z’ may or may not be included.

Arguments

`time`	date and time coded according to the CCSDS ASCII Calendar Segmented Time Code either format A or B

Result codes

`S_OK`	the time has been set
`E_INVALIDARG`	the argument does not contain a valid date and time representation
`E_FAIL`	failure to set the time because of other reasons

```
HRESULT Set_Time( const char* time );
```

Sets the time to the value specified by the input argument and the date to the current date. The ASCII string contains the time subset of the CCSDS ASCII Calendar Segmented Time Code. The time subset must contain at least the hours. The trailing 'Z' may or may not be included.

Arguments

`time`	time subset of the CCSDS ASCII Calendar Segmented Time Code

Result codes

`S_OK`	the time has been set
`E_INVALIDARG`	the argument does not contain a valid date and time representation
`E_FAIL`	failure to set the time because of other reasons

```
char* Get_Date( SLE_TimeFmt fmt ) const;
```

Returns an ASCII string with the date formatted according to the CCSDS ASCII Calendar Segmented Time Code in the variant specified by the input argument. The returned value must be deleted by the client.

Arguments

`fmt`	the variant of the time code to be used

```
char* Get_Time( SLE_TimeFmt fmt,
                SLE_TimeRes res = sleTR_seconds ) const;
```

Returns the time (no date) formatted according to the CCSDS ASCII Calendar Segmented Time Code in the variant and with the resolution specified by the input arguments. The optional 'Z' is included. The returned value must be deleted by the client.

Arguments

`fmt`	the variant of the time code to be used
`res`	the resolution of the time

```
char* Get_DateAndTime( SLE_TimeFmt fmt,
                       SLE_TimeRes res = sleTR_seconds ) const;
```

Returns the time and date formatted according to the CCSDS ASCII Calendar Segmented Time Code in the variant and with the resolution specified by the input arguments. The optional 'Z' is included. The returned value must be deleted by the client.

Arguments

`fmt`	the variant of the time code to be used
`res`	the resolution of the time

```
void Update();
```

Sets the value of the time to current time. Current time is obtained from the interface `ISLE_TimeSource`, if this interface was supplied to the creator-function of the component. Otherwise, the component uses system time.

```
double operator- (const ISLE_Time& time ) const;
```

Calculates the difference between the time stored and the time passed as argument and returns the difference measured in seconds and fractions of a second.

```
bool operator== ( const ISLE_Time& time ) const;
bool operator!= ( const ISLE_Time& time ) const;
bool operator<  ( const ISLE_Time& time ) const;
bool operator>  ( const ISLE_Time& time ) const;
bool operator<= ( const ISLE_Time& time ) const;
bool operator>= ( const ISLE_Time& time ) const;
```

Standard comparison operators for times.

```
ISLE_Time* Copy() const;
```

Copies the time object and returns the interface of the copy.

A4.5 SLE SERVICE INSTANCE IDENTIFIER

Name `ISLE_SII`
GUID `{EC5C1E4B-1E25-4280-AA17-BA8B510AEC20}`
Inheritance: `IUnknown`
File `ISLE_SII.H`

Objects exporting this interface handle the service instance identifier defined by the CCSDS Recommended Standards for SLE transfer services. The interface supports two formats for the service instance identifier:

a) The standard format as defined by reference [17] with the constraint that the attribute are always character strings.

b) A standard character string representation defined in the CCSDS Recommended Standards (for version 1 of the services RAF, RCF, and CLTU, this definition is provided by annex C of this specification).

The standard format consists of a sequence of 'attribute value assertions', i.e., pairs of an attribute identifier and an attribute value. The attribute identifier is an object identifier as defined by ASN.1 (reference [15]).

The object is able to process the standard ASCII representation for input and output.

It also accepts the standard format as defined by reference [17] and produces output in this format. For the global form of the object identifier the object uses the full object identifier presented as an array of integers. For the local form, it accepts and outputs only the trailing component of the object identifier, which is unique for all attributes used in a service instance identifier. For retrieval of the standard format, the object supports a simple built-in iterator by which the name components can be read.

The object verifies that the structure and contents of the service instance identifier conforms to the specifications provided in the CCSDS Recommended Standards for SLE transfer services (for version 1 to the specification in annex C).

After creation the value of the service instance identifier is NULL.

After creation, the format to be used is set to the one defined in the CCSDS Recommended Standards for SLE transfer services. To use the initial format defined in annex C, the method `Set_InitialFormat()` must be called. Support for the initial format is optional and implementations not supporting this format shall return an error when the method is called.

Synopsis

```
#include <SLE_SCM.H>

#define IID_ISLE_SII_DEF { 0xec5c1e4b, 0x1e25, 0x4280,  \
           { 0xaa, 0x17, 0xba, 0x8b, 0x51, 0xa, 0xec, 0x20 } }
```

```
interface ISLE_SII : IUnknown
{
  virtual bool
    Get_InitialFormatUsed() const = 0;
  virtual HRESULT
    Set_InitialFormat() = 0;
  virtual char*
    Get_AsciiForm() const = 0;
  virtual char*
    GetLastRDN() const = 0;
  virtual HRESULT
    Set_AsciiForm( const char* siiString ) = 0;
  virtual bool
    IsNull() const = 0;
  virtual void
    SetToNull() = 0;
  virtual bool
    operator== ( const ISLE_SII& sii ) const = 0;
  virtual bool
    operator!= (const ISLE_SII& sii ) const = 0;
  virtual ISLE_SII*
    Copy() const = 0;
  virtual HRESULT
    Add_GlobalRDN( const int objId[],
                   size_t objIdLength,
                   const char* value ) = 0;
  virtual HRESULT
    Add_LocalRDN( int objId, const char* value ) = 0;
  virtual void
    Reset() = 0;
  virtual bool
    MoreData() = 0;
  virtual HRESULT
    NextGlobalRDN( int*& objId,
                   size_t& objIdLength,
                   char*& value ) = 0;
  virtual HRESULT
    NextLocalRDN( int& objId,
                  char*& value ) = 0;
};
```

Methods

`Bool Get_InitialFormatUsed();`

Returns TRUE if the initial format defined in annex C is being used and FALSE otherwise.

`HRESULT Set_InitialFormat();`

Requests use of the initial format defined in annex C of this Specification to support version 1 of the services RAF, RCF, and CLTU.

Result codes

S_OK	the request has been accepted
E_NOTIMPL	the implementation does not support the initial format

char* Get_AsciiForm() const;

Returns the ASCII representation of the service instance identifier or the string '***' if the identifier is NULL. The string must be deleted by the client.

char* GetLastRDN() const;

Returns the ASCII representation of the last component of the service instance identifier or the string '***' if the identifier is NULL. The string must be deleted by the client.

HRESULT Set_AsciiForm(const char* siiString);

Parses the input string and sets the value of the service instance identifier as defined by the string. If the string is badly formatted or contains attributes that are not defined for the service instance identifier, returns an error and sets the value of the service instance identifier to NULL.

Arguments

siiString	an ASCII string defining the service instance identifier

Result codes

S_OK	the value of the service instance identifier has been set as defined by the input argument
E_INVALIDARG	syntax error in the input
SLE_E_INVALIDID	unknown attribute abbreviation encountered
SLE_E_SEQUENCE	incorrect sequence of attributes

bool IsNull() const;

Returns `true` if the value if the service instance identifier is NULL.

void SetToNull();

Sets the value of service instance identifier to NULL.

```
bool operator== ( const ISLE_SII& sii ) const;
bool operator!= ( const ISLE_SII& sii ) const;
```

The standard equality operators for service instance identifiers.

```
ISLE_SII* Copy() const;
```

Performs a deep copy of the service instance identifier and returns the copy.

```
HRESULT Add_GlobalRDN( const int objId[],
                       size_t objIdLength,
                       const char* value );
```

Appends a relative distinguished name (i.e., an attribute identifier - attribute value pair) to the end of the service instance identifier. If the object identifier is not defined for the service instance identifier or the value is empty, returns an error and does not add the name to service instance identifier.

Arguments

`objId`	the object identifier of the attribute presented as an array of integers; the array is copied by the object
`objIdLength`	the number of components of the object identifier
`value`	the value of the attribute

Result codes

`S_OK`	the relative distinguished name has been appended to the service instance identifier
`SLE_E_BADVALUE`	value contains characters that are not permitted
`SLE_E_INVALIDID`	unknown attribute identifier
`SLE_E_SEQUENCE`	incorrect sequence of attributes

```
HRESULT Add_LocalRDN( int objId, const char* value );
```

Appends a relative distinguished name to the end of the service instance identifier. For this method the attribute is identified by the last component of the object identifier. If the object identifier is not defined for the service instance identifier or the value is empty, returns an error and does not add the name to service instance identifier.

Arguments

`objId`	the last component of the object identifier of the attribute
`value`	the value of the attribute

Result codes

S_OK	the relative distinguished name has been appended to the service instance identifier
SLE_E_BADVALUE	value contains characters that are not permitted
SLE_E_INVALIDID	unknown attribute identifier
SLE_E_SEQUENCE	incorrect sequence of attributes

Iteration Methods

```
void Reset();
```

Resets the internal iterator to the beginning of the service instance identifier.

```
bool MoreData();
```

Returns `true` when the iterator has not yet reached the end of the service instance identifier. I.e., the next to call to `NextGlobalRDN()` or `NextLocalRDN()` will return a relative distinguished name. Otherwise, returns `false`.

```
HRESULT NextGlobalRDN( int*& objId,
                       size_t& objIdLength,
                       char*& value ) const;
```

Returns the relative distinguished name pointed at by the iterator in the global form and forwards the iterator by one element. If the iterator has reached the end of the service instance identifier or the service identifier is NULL, returns an error.

Arguments

objId	the object identifier of the attribute presented as an array of integers; the array is a copy of the internal data and must be deleted by the client
objIdLength	the length of the object identifier
value	the value of the attribute; this is a copy of the internal data, which must be deleted by the client

Result codes

S_OK	the output arguments contain the relative distinguished name
SLE_S_EOD	end of service instance identifier reached
SLE_S_NULL	service instance identifier is NULL

```
HRESULT NextLocalRDN( int& objId, char*& value ) const;
```

Returns the relative distinguished name pointed at by the iterator in the local form and forwards the iterator by one element. If the iterator has reached the end of the service instance identifier or the service identifier is NULL, returns an error.

Arguments

`objId`	the last component of the object identifier of the attribute
`value`	the value of the attribute; this is a copy of the internal data which must be deleted by the client

Result codes

`S_OK`	the output arguments contain the relative distinguished name
`SLE_S_EOD`	end of service instance identifier reached
`SLE_S_NULL`	service instance identifier is NULL

A4.6 SLE CREDENTIALS

Name ISLE_Credentials
GUID {D020B002-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_Credentials.H

Objects implementing this interface hold the credentials used for authentication of the peer identity. The credentials comprise a message digest (the protected), a random number, and the time when the message digest was generated. For the message digest the object does not make any assumptions on the format, size, or encoding. It simply stores the sequence of bytes passed to it.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_Time;

#define IID_ISLE_Credentials_DEF { 0xd020b002, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Credentials : IUnknown
{
  virtual long
    Get_RandomNumber() const = 0;
  virtual const SLE_Octet*
    Get_Protected( size_t& size ) const = 0;
  virtual const ISLE_Time&
    Get_TimeRef() const = 0;
  virtual void
    Set_RandomNumber( long number ) = 0;
  virtual void
    Set_Protected( const SLE_Octet* hashCode, size_t size ) = 0;
  virtual void
    Set_TimeRef( const ISLE_Time& time ) = 0;
  virtual bool
    operator== ( const ISLE_Credentials& credentials ) const = 0;
  virtual bool
    operator!= ( const ISLE_Credentials& credentials ) const = 0;
  virtual ISLE_Credentials*
    Copy() const = 0;
  virtual char*
    Dump() const = 0;
};
```

Methods

```
long Get_RandomNumber() const;
```

Returns the random number currently stored in the object or zero when no value has been set. Of course, zero is as random as any other number. A return value of zero does not indicate that the attribute has not been set.

```
const SLE_Octet* Get_Protected( size_t& size ) const;
```

Returns the hash code (‘the protected’) currently stored in the object or a NULL pointer when no value has been set.

```
const ISLE_Time& Get_TimeRef() const;
```

Returns the generation time stored in the object; if no values have been set, the time value is undefined.

```
void Set_RandomNumber( long number );
```

Sets the random number to the value of the input argument.

```
void Set_Protected( const SLE_Octet* hashCode, size_t size );
```

Sets the hash code in the object copying the input argument.

```
void Set_TimeRef( const ISLE_Time& time );
```

Sets the generation time in the object copying the input argument.

```
bool operator== ( const ISLE_Credentials& credentials ) const;
bool operator!= ( const ISLE_Credentials& credentials ) const;
```

Comparison operators for credentials.

```
ISLE_Credentials* Copy() const;
```

Performs a deep copy and returns it.

```
char* Dump() const;
```

Produces a human readable string of the object contents. The returned value must be deleted by the client.

A4.7 SLE SECURITY ATTRIBUTES

Name ISLE_SecAttributes
GUID {D020B003-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_SecAttributes.H

Objects implementing this interface hold the user name and the password required for generating and authenticating credentials. For the password the object does not make any assumptions on the format, size, or encoding. It simply stores the sequence of bytes passed to it.

The interface provides methods to generate credentials and to authenticate credentials. The procedure applied for both methods is specified in 3.5.6.

Because the object stores sensitive information it does not provide methods for read access and does not support printing of the contents.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_Credentials;

#define IID_ISLE_SecAttributes_DEF { 0xd020b003, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_SecAttributes : IUnknown
{
  virtual void
    Set_UserName( const char* name ) = 0;
  virtual void
    Set_Password( const SLE_Octet* pwd, size_t size ) = 0;
  virtual void
    Set_HexPassword( const char* pwd ) = 0;
  virtual ISLE_Credentials*
    GenerateCredentials() const = 0;
  virtual bool
    Authenticate( const ISLE_Credentials& credentials,
                  int acceptableDelay ) const = 0;
  virtual bool
    operator== ( const ISLE_SecAttributes& secAttr ) const = 0;
  virtual bool
    operator!= ( const ISLE_SecAttributes& secAttr ) const = 0;
  virtual ISLE_SecAttributes*
    Copy() const = 0;
};
```

Methods

`void Set_UserName( const char* name );`

Sets the user name to the value defined by the argument.

Arguments

`name`	user name

Precondition: The length of the string supplied as argument is >= 3 and <= 16 characters.

`void Set_Password( const SLE_Octet* pwd, size_t size );`

Sets the password to the value defined by the arguments.

Arguments

`pwd`	pointer to the password
`size`	the size of the password in bytes

Precondition: The length of the octet string supplied as argument is >= 6 and <= 16 octets.

`void Set_HexPassword( const char* pwd );`

Sets the password to the value described by the argument.

Arguments

`pwd`	An ASCII string with a hex representation of the password. The ASCII string consists of an even number of hex digits (two hex digits for each byte of the password) without blanks and without any prefix or postfix. The ASCII string may only contain ASCII characters from the set {‘0’ - ‘9’, ‘A’ - ‘F’, ‘a’ - ‘f’}.’

Precondition: The length of the octet string represented by the argument is >=6 and <=16 octets.

`ISLE_Credentials* GenerateCredentials() const;`

Generates and returns credentials from the attributes stored to the object. When any of the attributes have not been set, returns NULL.

`bool Authenticate( const ISLE_Credentials& credentials,`
`int acceptableDelay ) const;`

Verifies that the credentials passed as argument have been generated from the attributes stored to the object within the acceptable time delay. Returns `true`, if authentication succeeds and `false` otherwise.

Arguments

`credentials`	the credentials that shall be authenticated
`acceptableDelay`	the acceptable delay between the time the credentials have been generated and the time of authentication

```
bool operator== ( const ISLE_SecAttributes& secAttr ) const;
bool operator!= ( const ISLE_SecAttributes& secAttr ) const;
```

The standard equality operators for security attributes.

```
ISLE_SecAttributes* Copy() const;
```

Performs a deep copy of the object and returns the copy.

A5 SLE OPERATION OBJECTS

A5.1 COMPONENT CREATOR FUNCTION

File `<impl-id>.H`

The component implementing operation objects includes a function to obtain a pointer to the operation factory interface. The signature of this function is defined as:

```
extern "C" HRESULT
  <impl-id>_CreateOpFactory( const GUID& iid,
                             ISLE_UtilFactory* putil,
                             ISLE_Reporter* preporter,
                             void** ppv );
```

where `<impl-id>` is replaced by the product identifier of the implementation. Note that external 'C' linkage is required. A reference to the utility factory that shall be used by the component must be passed as an argument. The function checks the argument identifying the operation factory interface and returns an error when the implementation does not support that identifier.

A pointer to the reporter interface can optionally be passed as well. Operation objects may use this interface to report error messages, if it is provided. The extent to which error logging is supported is implementation dependent.

Arguments

`iid`	identifier of the required interface
`putil`	pointer to the interface of the Utility Factory
`preporter`	pointer to the reporter interface for passing of log messages and notifications to the application
`ppv`	pointer to the requested interface of the Operation Factory

Result codes

`S_OK`	the object has been created
`E_NOINTERFACE`	the specified interface is not supported
`E_INVALIDARG`	the reference to the utility factory or reporter is missing

A5.2 SLE OPERATION FACTORY

Name ISLE_OperationFactory
GUID {BB4DDA22-54CD-11d8-9CF5-0004761E8CFB}
Inheritance: IUnknown
File ISLE_OperationFactory

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
#include <SLE_Types.h>
interface ISLE_Operation;

#define IID_ISLE_OperationFactory_DEF { 0xbb4dda22, 0x54cd, 0x11d8, \
         { 0x9c, 0xf5, 0x0, 0x4, 0x76, 0x1e, 0x8c, 0xfb } }

interface ISLE_OperationFactory : IUnknown
{
  virtual HRESULT
    CreateOperation( const GUID& iid,
                     SLE_OpType opType,
                     SLE_ApplicationIdentifier srvType,
                     SLE_VersionNumber version,
                     void** ppv ) const = 0;
};
```

Methods

```
HRESULT CreateOperation( const GUID& iid,
                         SLE_OpType opType,
                         SLE_ApplicationIdentifier srvType,
                         SLE_VersionNumber version,
                         void ** ppv ) const;
```

Creates a new operation object as specified by the arguments. If the interface cannot be found, does not refer to an operation object interface of the specified type, or is not supported for the specified service type, returns an error and sets the output argument to NULL.

Arguments

`iid`	GUID for the operation object interface to be returned
`opType`	the operation object type that shall be created
`srvType`	the service type for which an operation is requested
`version`	the version number of the service type identified by `srvType`, which must be greater than zero
`ppv`	a pointer to the requested interface

Result codes

S_OK	the object has been created
E_NOINTERFACE	the specified interface is not supported
SLE_E_INCONSISTENT	the requested operation type is not supported by the specified service type

A5.3 BASIC INTERFACES

A5.3.1 SLE Operation

Name ISLE_Operation
GUID {BB4DDA25-54CD-11d8-9CF5-0004761E8CFB}
Inheritance: IUnknown
File ISLE_Operation.H

The interface defines basic characteristics supported by all operation objects.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
#include <SLE_Types.h>
interface ISLE_Credentials;

#define IID_ISLE_Operation_DEF { 0xbb4dda25, 0x54cd, 0x11d8, \
          { 0x9c, 0xf5, 0x0, 0x4, 0x76, 0x1e, 0x8c, 0xfb } }

interface ISLE_Operation : IUnknown
{
  virtual SLE_ApplicationIdentifier
    Get_OpServiceType() const = 0;
  virtual SLE_VersionNumber
    Get_OpVersionNumber() const = 0;
  virtual SLE_OpType
    Get_OperationType() const = 0;
  virtual bool
    IsConfirmed() const = 0;
  virtual const ISLE_Credentials*
    Get_InvokerCredentials() const = 0;
  virtual void
    Set_InvokerCredentials( const ISLE_Credentials& credentials ) = 0;
  virtual void
    Put_InvokerCredentials( ISLE_Credentials* pcredentials ) = 0;
  virtual HRESULT
    VerifyInvocationArguments() const = 0;
  virtual HRESULT
    Lock() = 0;
  virtual HRESULT
    TryLock() = 0;
  virtual HRESULT
    Unlock() = 0;
  virtual ISLE_Operation*
    Copy()const = 0;
  virtual char*
    Print( int maxDumpLength ) const = 0;
};
```

Methods

`SLE_ApplicationIdentifier Get_OpServiceType() const;`

Returns the SLE service type for the operation.

`SLE_VersionNumber Get_OpVersionNumber() const;`

Returns the version number of the SLE Service type for the operation.

`SLE_OpType Get_OperationType() const;`

Returns the type of operation as defined by `SLE_OpType`.

`bool IsConfirmed() const;`

Returns true if the operation is a confirmed operation, false otherwise.

`const ISLE_Credentials* Get_InvokerCredentials() const;`

Returns a pointer to the invoker credentials or NULL when no credentials are present.

`void Set_InvokerCredentials( const ISLE_Credentials& credentials );`

Sets the invoker credentials copying the input argument.

`void Put_InvokerCredentials( ISLE_Credentials* pcredentials );`

Sets the invoker credentials to the input argument. The input argument will be deleted by the operation object.

`HRESULT VerifyInvocationArguments() const;`

Verifies the invocation arguments with respect to completeness, consistency, and range.

Result codes

S_OK	all checks passed
SLE_E_MISSINGARG	at least one argument is missing
SLE_E_INCONSISTENT	the arguments are inconsistent
SLE_E_RANGE	at least one argument is out of range

Further results codes might be specified by supplemental Recommended Practice documents for service-specific APIs. These result codes must be taken into account by implementations of derived interfaces.

`HRESULT Lock();`

Sets an advisory lock on the operation object.

Result codes

S_OK	lock has been set
E_FAIL	further unspecified error

`HRESULT TryLock();`

Sets the lock on the object if possible. If the lock is currently not available returns immediately.

Result codes

S_OK	lock has been set
SLE_S_LOCKED	lock not available
E_FAIL	further unspecified error

`HRESULT Unlock();`

Releases a lock previously set on the operation object.

Result codes

S_OK	lock has been removed
E_FAIL	further unspecified error

`ISLE_Operation* Copy();`

Performs a deep copy of the operation object and returns a pointer to the new object.

`char* Print( int maxDumpLength ) const;`

Generates and returns a human readable printout of the object attributes.

Arguments

`maxDumpLength` defines the maximum length in bytes of hexadecimal dumps produced for binary parameters, such as space data units

Default Setting of Operation Parameters after Creation

Argument	Created directly	Created by Service Instance
service type	set in the create request	as defined for the SI
version number	set in the create request	as defined for the SI
operation type	set in the create request	according to the request
confirmed operation	depending on derived I/F	depending on derived I/F
invoker credentials used	false	false
invoker credentials	NULL	NULL

Checking of Invocation Parameters

No checks are defined for the parameters handled by this interface.

A5.3.2 SLE Confirmed Operation

Name ISLE_ConfirmedOperation
GUID {D020B006-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown – ISLE_Operation
File ISLE_ConfirmedOperation.H

The interface defines characteristics supported by all confirmed operation objects.

Synopsis

```
#include <ISLE_Operation.H>

#define IID_ISLE_ConfirmedOperation_DEF {0xd020b006, 0xccd1, 0x11d2,\
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_ConfirmedOperation : ISLE_Operation
{
  virtual SLE_Result
    Get_Result() const = 0;
  virtual SLE_DiagnosticType
    Get_DiagnosticType() const = 0;
  virtual SLE_Diagnostics
    Get_Diagnostics() const = 0;
  virtual SLE_InvokeId
    Get_InvokeId() const = 0;
  virtual const ISLE_Credentials*
    Get_PerformerCredentials() const = 0;
  virtual void
    Set_PositiveResult() = 0;
  virtual void
    Set_Diagnostics( SLE_Diagnostics diagnostic ) = 0;
  virtual void
    Set_InvokeId( SLE_InvokeId id ) = 0;
  virtual void
    Set_PerformerCredentials( const ISLE_Credentials& credentials ) = 0;
  virtual void
    Put_PerformerCredentials( ISLE_Credentials* pcredentials ) = 0;
  virtual HRESULT
    VerifyReturnArguments() const = 0;
};
```

Methods

`SLE_Result Get_Result() const;`

Returns the result (positive / negative) stored to the object.

`SLE_DiagnosticType Get_DiagnosticType() const;`

Returns the type of diagnostic (general, special, none) stored to the object.

`SLE_Diagnostics Get_Diagnostics() const;`

Returns the common diagnostics if set in the object.

`SLE_InvokeId Get_InvokeId() const;`

Returns the Invocation Identifier currently set in the object.

`const ISLE_Credentials* Get_PerformerCredentials() const;`

Returns a pointer to the performer credentials or NULL when no credentials are present.

`void Set_PositiveResult();`

Sets the result of the operation to positive and the diagnostic type to ‘invalid’. A negative result is set with the diagnostics.

`void Set_Diagnostics( SLE_Diagnostics diagnostic );`

Sets the common diagnostics to the input argument, sets the diagnostic type to ‘common’ and the result to negative.

`void Set_InvokeId( SLE_InvokeId id );`

Sets the invoke identifier to the value passed as argument.

`void Set_PerformerCredentials( const ISLE_Credentials& credentials );`

Sets the performer credentials copying the input argument.

`void Put_PerformerCredentials( ISLE_Credentials* pcredentials );`

Sets the performer credentials to the input argument. The credentials argument will be deleted by the operation object.

`HRESULT VerifyReturnArguments();`

Verifies the invocation arguments with respect to completeness, consistency, and range.

Result codes

S_OK	all checks passed
SLE_E_MISSINGARG	at least one argument is missing
SLE_E_INCONSISTENT	the arguments are inconsistent, e.g., do not match the invocation arguments
SLE_E_RANGE	at least one argument is out of range
SLE_E_DIAGNOSTIC	the diagnostic code is missing, unknown, or inconsistent with the result

Further results codes might be specified by for specific SLE services. These result codes must be taken into account by implementations of derived interfaces.

Default Setting of Operation Parameters after Creation

Argument	Created directly	Created by Service Instance
result	'invalid'	'invalid'
diagnostic type	'none'	'none'
common diagnostics	'invalid'	'invalid'
performer credentials used	false	false
performer credentials	NULL	NULL
invocation identifier	0	0 (will be handled by the service instance)

Checking of Return Parameters

Argument	Required condition
result	must be set
diagnostic type	if the result is 'negative' must be 'common' or 'specific'
common diagnostics	if the diagnostic type is common, must not be 'invalid'

A5.4 COMMON ASSOCIATION MANAGEMENT

A5.4.1 BIND Operation

Name `ISLE_Bind`
GUID `{D020B007-CCD1-11d2-9B44-00A0246D80DB}`
Inheritance: `IUnknown – ISLE_Operation – ISLE_ConfirmedOperation`
File `ISLE_Bind.H`

The interface provides access to the parameters of the operation BIND. Through its inheritance, it provides access to the parameter 'invocation identifier'. This parameter is not defined for the BIND operation and must not be used. The API proxy and the API Service Element must exclude this operation from the checks related to invocation identifiers.

The SLE service type and version number applicable for the operation object are defined when the object is created (see `ISLE_UtilFactory`) and the interface `ISLE_Operation` provides access to these attributes. This interface defines additional attributes for the SLE service type and version number, which represent the parameters of the SLE BIND operation and can be modified via this interface.

Synopsis

```
#include <ISLE_ConfirmedOperation.H>
interface ISLE_SII;

#define IID_ISLE_Bind_DEF { 0xd020b007, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Bind : ISLE_ConfirmedOperation
{
  virtual const char*
    Get_InitiatorIdentifier() const = 0;
  virtual const char*
    Get_ResponderIdentifier() const = 0;
  virtual const char*
    Get_ResponderPortIdentifier() const = 0;
  virtual const ISLE_SII&
    Get_ServiceInstanceId() const = 0;
  virtual void
    Set_InitiatorIdentifier( const char* id ) = 0;
  virtual void
    Set_ResponderIdentifier( const char* id ) = 0;
  virtual void
    Set_ResponderPortIdentifier( const char* port ) = 0;
  virtual void
    Set_ServiceInstanceId( const ISLE_SII& siid ) = 0;
  virtual void
    Put_ServiceInstanceId( ISLE_SII* psiid ) = 0;
  virtual SLE_ApplicationIdentifier
    Get_ServiceType() const = 0;
  virtual SLE_VersionNumber
    Get_VersionNumber() const = 0;
```

```
  virtual void
    Set_ServiceType( SLE_ApplicationIdentifier serviceType ) = 0;
  virtual void
    Set_VersionNumber( SLE_VersionNumber version ) = 0;
  virtual SLE_BindDiagnostic
    Get_BindDiagnostic() const = 0;
  virtual void
    Set_BindDiagnostic( SLE_BindDiagnostic diagnostic ) = 0;
};
```

Methods

`const char* Get_InitiatorIdentifier() const;`

Returns the identifier for initiating SLE Application set in the object or NULL if the parameter has not been set.

`const char* Get_ResponderIdentifier() const;`

Returns the identifier for responding SLE Application set in the object or NULL if the parameter has not been set.

`const char* Get_ResponderPortIdentifier() const;`

Returns the responder port identifier if currently set in the object. Otherwise returns a NULL pointer.

`const ISLE_SII& Get_ServiceInstanceId() const;`

Returns the service instance identifier set in the object.

Precondition: service instance identifier is present in the object.

`void Set_InitiatorIdentifier( const char* id );`

Sets the identifier of the initiating SLE Application.

`void Set_ResponderIdentifier( const char* id );`

Sets the identifier of the responding SLE Application.

`void Set_ResponderPortIdentifier( const char* port );`

Sets the initiator port identifier copying the input argument.

`void Set_ServiceInstanceId( const ISLE_SII& siid );`

Sets the service instance identifier copying the input argument.

`void Put_ServiceInstanceId( ISLE_SII* psiid );`

Sets the service instance identifier to the input argument. The service instance identifier passed is deleted by the component.

`SLE_ApplicationIdentifier Get_ServiceType();`

Returns the service type currently stored in the object. Note that this method differs from the inherited method `Get_OpServiceType()`, which returns the service type for which the operation object was created (see `ISLE_UtilFactory` and `ISLE_Operation`).

`SLE_VersionNumber Get_VersionNumber() const;`

Returns the version number currently set in the object. Note that this method differs from the inherited method `Get_OpVersionNumber()`, which returns the version of the service type for which the operation object was created (see `ISLE_UtilFactory` and `ISLE_Operation`).

`void Set_ServiceType( SLE_ApplicationIdentifier serviceType );`

Sets the service type to the input argument.

`void Set_VersionNumber( SLE_VersionNumber version );`

Sets the version to the input argument.

`SLE_BindDiagnostic Get_BindDiagnostic();`

Returns the bind diagnostic currently set in the object.

`void Set_BindDiagnostic( SLE_BindDiagnostic diagnostic );`

Sets the result to negative, the diagnostic type to ‘specific’, and the diagnostics to the input argument.

Default Setting of Operation Parameters after Creation

Argument	Created directly	Created by Service Instance
initiator identifier	NULL	NULL
responder identifier	NULL	as defined for the SI
responder port identifier	NULL	as defined for the SI
service instance identifier	NULL	as defined for the SI
service type	as requested for creation	as defined for the SI
version number	as requested for creation	as defined for the SI
bind diagnostic	‘invalid’	‘invalid’

Checking of Invocation Parameters

Argument	Required condition
responder identifier	must be set
responder port identifier	must be set
service instance identifier	must be set
service type	must be set
version number	must be set

The checks defined here include a check on the presence of the responder identifier and therefore apply to the BIND initiating side only. On the responding side, the method `VerifyInvocationArguments()` should not be called, because it might return an error although the BIND invocation PDU is correct. Calling of this method on the responder side is not necessary, because all arguments are subject to specific tests performed by the components API proxy and API Service Element.

Checking of Return Parameters

Argument	Required condition
bind diagnostic	if the result is negative and the diagnostic type is ‘specific’ must not be ‘invalid’

A5.4.2 UNBIND Operation

Name ISLE_Unbind
GUID {7B425720-D32D-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown – ISLE_Operation – ISLE_ConfirmedOperation
File ISLE_Unbind.H

The interface provides access to the parameters of the operation UNBIND. Through its inheritance, it provides access to the parameter 'invocation identifier'. This parameter is not defined for the UNBIND operation and must not be used. The API proxy and the API Service Element must exclude this operation from the checks related to invocation identifiers.

Synopsis

```
#include <ISLE_ConfirmedOperation.H>

#define IID_ISLE_Unbind_DEF { 0x7b425720, 0xd32d, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Unbind : ISLE_ConfirmedOperation
{
  virtual SLE_UnbindReason
    Get_UnbindReason() const = 0;
  virtual void
    Set_UnbindReason( SLE_UnbindReason reason ) = 0;
};
```

Methods

`SLE_UnbindReason Get_UnbindReason() const;`

Returns the unbind reason currently set in the object.

`void Set_UnbindReason( SLE_UnbindReason reason );`

Sets the unbind reason to the input argument.

Default Setting of Operation Parameters after Creation

Argument	Created directly	Created by Service Instance
unbind reason	'invalid'	'invalid'

Checking of Invocation Parameters

Argument	Required condition
unbind reason	must not be 'invalid'

Checking of Return Parameters

No checks are defined for parameters handled by this interface.

A5.4.3 PEER-ABORT Operation

Name ISLE_PeerAbort
GUID {7B425721-D32D-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown – ISLE_Operation
File ISLE_PeerAbort.H

The interface provides access to the parameters of the operation PEER-ABORT. Through its inheritance, it provides access to the parameter 'invoker credentials'. This parameter is not defined for the PEER-ABORT operation and must not be used. The proxy must ensure that authentication is not applied to the PEER-ABORT operation, even if the parameter is set in the operation object by mistake.

In addition to the parameters defined for the SLE operation, objects exporting this interface store the originator of the abort, which can be the peer system, the local proxy, the local service element, or the local application. This information is not forwarded across the association.

Synopsis

```
#include <ISLE_Operation.H>

#define IID_ISLE_PeerAbort_DEF { 0x7b425721, 0xd32d, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_PeerAbort : ISLE_Operation
{
  virtual SLE_PeerAbortDiagnostic
    Get_PeerAbortDiagnostic() const = 0;
  virtual void
    Set_PeerAbortDiagnostic( SLE_PeerAbortDiagnostic diagnostic ) = 0;
  virtual SLE_AbortOriginator
    Get_AbortOriginator() const = 0;
  virtual void
    Set_AbortOriginator( SLE_AbortOriginator originator ) = 0;
};
```

Methods

`SLE_PeerAbortDiagnostic Get_PeerAbortDiagnostic() const;`

Returns the special diagnostics for PEER-ABORT, if these are available. The type of the diagnostics can be checked by the method provided by the base class.

`void Set_PeerAbortDiagnostic( SLE_PeerAbortDiagnostic diagnostic );`

Sets the PEER-ABORT diagnostic, and the diagnostic type to 'specific'.

`SLE_AbortOriginator Get_AbortOriginator() const;`

Returns the originator of the abort.

```
void Set_AbortOriginator( SLE_AbortOriginator originator );
```

Sets the originator of the abort to the input argument.

Default Setting of Parameters after Creation

Argument	Created directly	Created by Service Instance
peer abort diagnostics	'invalid'	'invalid'
abort originator	'invalid'	'application'

Checking of Invocation Parameters

No checking is performed.

A5.5 OTHER COMMON OPERATIONS

A5.5.1 STOP Operation

Name ISLE_Stop
GUID {7B425723-D32D-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown – ISLE_Operation – ISLE_ConfirmedOperation
File ISLE_Stop.H

This is an empty interface, as all functionality required is covered by the inherited interfaces. The specific interface exists in order to attach the required identifier.

Synopsis

```
#include <ISLE_ConfirmedOperation.H>

#define IID_ISLE_Stop_DEF { 0x7b425723, 0xd32d, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Stop : ISLE_ConfirmedOperation {};
```

Methods

This interface does not define any new methods.

A5.5.2 SCHEDULE STATUS REPORT Operation

Name ISLE_ScheduleStatusReport
GUID {7B425724-D32D-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown – ISLE_Operation – ISLE_ConfirmedOperation
File ISLE_ScheduleStatusReport.H

Synopsis

```
#include <ISLE_ConfirmedOperation.H>

#define IID_ISLE_ScheduleStatusReport_DEF \
          { 0x7b425724, 0xd32d, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_ScheduleStatusReport : ISLE_ConfirmedOperation
{
  virtual SLE_ReportRequestType
    Get_ReportRequestType() const = 0;
  virtual void
    Set_ReportRequestType( SLE_ReportRequestType type ) = 0;
  virtual SLE_ReportingCycle
    Get_ReportingCycle() const = 0;
  virtual void
    Set_ReportingCycle( SLE_ReportingCycle cycle ) = 0;
  virtual SLE_ScheduleStatusReportDiagnostic
    Get_SSRDiagnostic() const = 0;
  virtual void
    Set_SSRDiagnostic( SLE_ScheduleStatusReportDiagnostic diagnostic )= 0;
};
```

Methods

`SLE_ReportRequestType Get_ReportRequestType() const;`

Returns the type of request (immediate, periodically, stop).

`void Set_ReportRequestType( SLE_ReportRequestType type );`

Sets the type of request.

`SLE_ReportingCycle Get_ReportingCycle() const;`

Returns the reporting cycle value currently set in the object.

Precondition: The report request type is set to ‘periodically’.

`void Set_ReportingCycle( SLE_ReportingCycle cycle );`

Sets the reporting cycle to the value passed as argument.

Precondition: The report request type is set to ‘periodically’.

```
SLE_ScheduleStatusReportDiagnostic Get_SSRDiagnostic() const;
```

Returns the diagnostic code if set in the object.

```
void Set_SSRDiagnostic( SLE_ScheduleStatusReportDiagnostic
diagnostic );
```

Sets the diagnostic code to the value of the argument, the diagnostic type to ‘specific’, and the result to ‘negative’.

Default Setting of Operation Parameters after Creation

Argument	Created directly	Created by Service Instance
report request type	‘invalid’	‘invalid’
reporting cycle	zero	zero
schedule status report diagnostic	‘invalid’	‘invalid’

Checking of Invocation Parameters

Argument	Required condition
report request type	must not be ‘invalid’
reporting cycle	must be set if the reporting type is periodically; if used the value must be in the range 2 to 600 seconds

Checking of Return Parameters

Argument	Required condition
schedule status report diagnostic	if the diagnostic type is ‘specific’ must not be ‘invalid’

A5.5.3 TRANSFER BUFFER Operation

Name ISLE_TransferBuffer
GUID {7B425725-D32D-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown – ISLE_Operation
File ISLE_TransferBuffer.H

Synopsis

```
#include <ISLE_Operation.H>

#define IID_ISLE_TransferBuffer_DEF { 0x7b425725, 0xd32d, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_TransferBuffer : ISLE_Operation
{
  virtual size_t
    Get_MaximumSize() const = 0;
  virtual HRESULT
    Set_MaximumSize( size_t size ) = 0;
  virtual size_t
    Get_Size() const = 0;
  virtual bool
    Full() const = 0;
  virtual bool
    Empty() const = 0;
  virtual void
    Append( ISLE_Operation* poperation ) = 0;
  virtual void
    Prepend( ISLE_Operation* poperation,
             bool extend = false ) = 0;
  virtual ISLE_Operation*
    RemoveFront() = 0;
  virtual ISLE_Operation*
    RemoveRear() = 0;
  virtual const ISLE_Operation*
    Front() const = 0;
  virtual void
    Clear() = 0;
  virtual void
    Reset() = 0;
  virtual bool
    MoreData() const = 0;
  virtual const ISLE_Operation*
    Next() = 0;
};
```

Methods

`size_t Get_MaximumSize() const;`

Returns the maximum number of elements that can be stored into the buffer.

`HRESULT Set_MaximumSize( size_t size );`

Sets the maximum number of elements that can be stored into the buffer. If the current size exceeds the requested maximum size returns an error.

Result codes

`S_OK`	the maximum size has been set as requested
`E_FAIL`	the maximum size has not been set because it would require deletion of stored objects

`size_t Get_Size() const;`

Returns the number of elements currently stored in the buffer.

`bool Full() const;`

Returns true if the number of stored elements equals the maximum number that can be stored.

`bool Empty() const;`

Returns true if nothing is stored in the buffer.

`void Append( ISLE_Operation* poperation );`

Appends the operation object to the end of the buffer. The operation object will be deleted by the buffer when it itself is deleted.

Preconditions: The buffer is not full and the operation object is of the correct type.

`void Prepend( ISLE_Operation* poperation, bool extend = false );`

Inserts the operation object at the front of the buffer. If the argument ‘extend’ is set to true, the buffer is extended if it is already full and the maximum size is adjusted.

Preconditions: The buffer is not full or the argument ‘extend’ is set to true; the operation object is of the correct type.

`ISLE_Operation* RemoveFront();`

Returns the operation object at the beginning of the buffer and removes it from the buffer. If the buffer is empty returns NULL.

`ISLE_Operation* RemoveRear();`

Returns the operation object at the end of the buffer and removes it from the buffer. If the buffer is empty returns NULL.

`const ISLE_Operation* Front() const;`

Returns a pointer to the first object in the buffer, without changing the buffer content.

`void Clear();`

Remove and delete all stored objects.

Iterating through the transfer buffer.

The following methods define a simple iterator for the transfer buffer. Iteration is always from the first to the last element stored.

`void Reset();`

Resets the iterator to the beginning of the buffer.

`bool MoreData() const;`

Returns true if more objects are stored in the buffer; i.e., the next call to `Next()` will return an object. If the iterator has reached the end of the buffer, returns false.

`const ISLE_Operation* Next();`

Returns the object at the position of the iterator and advances the iterator by one.

Code Example for iteration through the buffer (pbuf is a pointer to the buffer):

```
const ISLE_Operation* poperation = 0;
pbuf->Reset();
while ( buf->MoreData() ) {
   poperation = buf->Next();
   // do something with the object
}
```

Default Setting of Operation Parameters after Creation

Argument	Created directly
maximum buffer size	1
current size	0

Checking of Invocation Parameters

No checks are defined.

A6 INTERFACES PROVIDED BY SEVERAL COMPONENTS

A6.1 CONTROL OF INTERFACE BEHAVIOR

A6.1.1 Sequential Flows of Control

Name `ISLE_Sequential`
GUID `{D020B008-CCD1-11d2-9B44-00A0246D80DB}`
Inheritance: `IUnknown`
File `ISLE_Sequential.H`

The interface is used to control processing of a component providing the behavior 'Sequential Flows of Control' as defined in 3.7.2.

Processing of the component is started with the method `StartSequential()` and stopped by `TerminateSequential()`. `StartSequential()` returns as soon as processing of the component has started.

The event monitor (interface `ISLE_EventMonitor`) is used by the component to register events on which the component implementing this interface will wait. The timer handler (interface `ISLE_TimerHandler`) is used by the component to start timers and register a timeout processor to be called when the timer expires.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_EventMonitor;
interface ISLE_TimerHandler;

#define IID_ISLE_Sequential_DEF { 0xd020b008, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Sequential : IUnknown
{
  virtual HRESULT
    StartSequential( ISLE_EventMonitor* pmonitor,
                     ISLE_TimerHandler* ptimerhandler ) = 0;
  virtual HRESULT
    TerminateSequential() = 0;
};
```

Methods

`HRESULT StartSequential( ISLE_EventMonitor* pmonitor,`
`                         ISLE_TimerHandler* ptimerhandler );`

Starts processing of the component.

Arguments

`pmonitor`	reference to the event monitor the component shall use for monitoring external events
`ptimerhandler`	reference to the timer handler the component shall use

Result codes

`S_OK`	processing of the component has started
`SLE_E_DEGRADED`	not all of the proxies linked to the service element could be started (applies only for the service element)
`SLE_E_CONFIG`	configuration has not been performed or has not completed successfully
`E_INVALIDARG`	either the event monitor or the timer handler are missing
`SLE_E_STATE`	operation of the component has already been started
`E_FAIL`	operation could not be started because of any other problem

`HRESULT TerminateSequential();`

Terminates processing of the component.

Result codes

`S_OK`	processing of the component will terminate
`SLE_E_STATE`	operation of the component has not been started

A6.1.2 Event Monitor

Name ISLE_EventMonitor
GUID {D020B009-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_EventMonitor.H

Objects implementing this interface provide the means to register and de-register external events, which the object will monitor. When an event is detected, the object will call the method ProcessEvent() of the interface ISLE_EventProcessor passed as argument to the event registration method. If, for any reason the object is no longer able to monitor an event, it calls the method MonitorAbort() of the event processor.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_EventProcessor;

#define IID_ISLE_EventMonitor_DEF { 0xd020b009, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_EventMonitor : IUnknown
{
  virtual HRESULT
    AddEvent( const SLE_EventHandle& handle,
              ISLE_EventProcessor* pprocessor ) = 0;
  virtual HRESULT
    RemoveEvent( const SLE_EventHandle& handle ) = 0;
};
```

Methods

`HRESULT AddEvent( const SLE_EventHandle& handle,`
`                  ISLE_EventProcessor* pprocessor );`

Registers the event identified by the event handle and the event processor that will process the event.

Arguments

handle	the event handle, describing the event according to platform specific conventions
pprocessor	pointer to the interface of the event processor that shall be invoked when the event is detected

Result codes

S_OK	the event has been registered
SLE_E_OVERFLOW	the number of registered events exceeds the capabilities of the event monitor
SLE_E_DUPLICATE	the event is already registered
E_FAIL	the request fails because of a further unspecified error

`HRESULT RemoveEvent( const SLE_EventHandle& handle );`

Removes a previously registered event and its event handler from the event monitor.

Arguments

handle	the event handle, describing the event according to platform specific conventions

Result codes

S_OK	the event has been de-registered
SLE_E_UNKNOWN	the event is not registered

A6.1.3 Event Processor

Name ISLE_EventProcessor
GUID {D020B00A-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_EventProcessor.H

The event processor handles an event detected by the event monitor with which it has been registered.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

#define IID_ISLE_EventProcessor_DEF { 0xd020b00a, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_EventProcessor : IUnknown
{
  virtual void
    ProcessEvent( const SLE_EventHandle& handle ) = 0;
  virtual void
    MonitorAbort( const SLE_EventHandle& handle ) = 0;
};
```

Methods

`void ProcessEvent( const SLE_EventHandle& handle );`

Processes the event passed as argument.

Arguments

`handle`	the event handle describing the event that has occurred

`void MonitorAbort( const SLE_EventHandle& handle );`

The method is called when the event handler is no longer able to monitor the event.

Arguments

`handle`	the event handle that had been registered

A6.1.4 Timer Handler

Name `ISLE_TimerHandler`
GUID `{0E265180-D4BF-11d2-9B44-00A0246D80DB}`
Inheritance: `IUnknown`
File `ISLE_TimerHandler.H`

Objects implementing this interface provide the means to start a timer and register a timeout processor. When the timer expires, the method `ProcessTimeout()` of the timeout processor is called. The interface also provides a method to cancel a running timer and to restart a timer that is already running. If for any reason the timer handler aborts a running timer by itself it calls the method `HandlerAbort()` of the timeout processor.

A running timer is identified by a timer identifier. This is an opaque type, with which the client must not associate any specific meaning. A specific identifier is only valid as long as the associated timer is running.

As an option, an invocation identifier can be associated with every activation of a timer. This invocation identifier is passed to the matching call of the method `ProcessTimeout`() of the timeout processor.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_TimeoutProcessor;


#define IID_ISLE_TimerHandler_DEF { 0xe265180, 0xd4bf, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } };

interface ISLE_TimerHandler : IUnknown
{
  virtual HRESULT
    StartTimer( int timeout,
                ISLE_TimeoutProcessor* pprocessor,
                SLE_TimerId& timer,
                int invocationId = 0 ) = 0;
  virtual HRESULT
    CancelTimer( SLE_TimerId timer ) = 0;
  virtual HRESULT
    RestartTimer( SLE_TimerId timer,
                  int timeout,
                  int invocationId = 0 ) = 0;
};
```

Methods

```
HRESULT StartTimer( int timeout,
                    ISLE_TimeoutProcessor* pprocessor,
                    SLE_TimerId& timer,
                    int invocationId );
```

Starts a timer and registers a timeout processor to be called when the timer expires.

Arguments

`timeout`	the timeout value in seconds
`pprocessor`	pointer to the interface of the timeout processor that shall be invoked when the timer expires
`timer`	the identifier for the timer returned to the caller
`invocationId`	identifier of the timer activation passed to the matching call of the timeout processor

Result codes

`S_OK`	the timer has been started
`SLE_E_OVERFLOW`	the number of timers exceeds the capabilities of the timer handler
`SLE_E_TIME`	the time specified cannot be handled
`E_FAIL`	the request fails because of a further unspecified error

```
HRESULT CancelTimer( SLE_TimerId timer );
```

Cancels a previously started timer.

Arguments

`timer`	the timer id returned from the call to `StartTimer()`

Result codes

`S_OK`	the timer has been cancelled
`SLE_E_UNKNOWN`	the timer is not running

```
HRESULT RestartTimer( SLE_TimerId timer, int timeout,
                      int invocationId );
```

Cancels and subsequently starts the timer identified in the first argument. Returns an error if the timer is not active.

Arguments

`timer`	the timer id returned from the call to `StartTimer()`
`timeout`	the timeout value in seconds
`invocationId`	identifier of the timer activation passed to the matching call of the timeout processor

Result codes

S_OK	the timer has been restarted
SLE_E_UNKNOWN	the timer is not active
SLE_E_TIME	the time specified cannot be handled
E_FAIL	the request fails because of a further unspecified reason

A6.1.5 Timeout Processor

Name ISLE_TimeoutProcessor
GUID {0E265181-D4BF-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_TimeoutProcessor.H

The timeout processor is called when a timer expires.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

#define IID_ISLE_TimeoutProcessor_DEF { 0xe265181, 0xd4bf, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } };

interface ISLE_TimeoutProcessor : IUnknown
{
  virtual void
    ProcessTimeout( SLE_TimerId timer,
                    int invocationId ) = 0;
  virtual void
    HandlerAbort( SLE_TimerId timer ) = 0;
};
```

Methods

`void ProcessTimeout( SLE_TimerId timer, int invocationId );`

Processes a timeout.

Arguments

timer	the timer id returned when the timer was started
invocationId	identifier of the timer activation passed to the call of the interface ISLE_TimerHandler, which caused this method invocation

`void HandlerAbort( SLE_TimerId timer );`

The method is called when the timer handler has aborted the timer for whatever reason.

Arguments

`timer`	the timer id returned when the timer was started

A6.1.6 Concurrent Flows of Control

Name `ISLE_Concurrent`
GUID `{7B425726-D32D-11d2-9B44-00A0246D80DB}`
Inheritance: `IUnknown`
File `ISLE_Concurrent.H`

The interface is used to control processing of a component providing the behavior ‘Concurrent Flows of Control’ as defined in 3.7.3.

Processing of the component is started with the method `StartConcurent()`. The method checks the configuration and returns as soon as processing within the component has been started.

Synopsis

```
#include <SLE_SCM.H>

#define IID_ISLE_Concurrent_DEF { 0x7b425726, 0xd32d, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Concurrent : IUnknown
{
  virtual HRESULT
    StartConcurrent() = 0;
  virtual HRESULT
    TerminateConcurrent() = 0;
};
```

Methods

`HRESULT StartConcurrent();`

Starts processing of the component.

Result codes

S_OK	processing of the component has started
SLE_E_DEGRADED	not all of the proxies linked to the service element could be started (applies only for the service element)
SLE_E_CONFIG	configuration has not been performed or has not completed successfully
SLE_E_STATE	operation of the component has already been started
E_FAIL	operation could not be started because of any other problem

`HRESULT TerminateConcurrent();`

Terminates processing of the component.

Result codes

S_OK	processing of the component will terminate
SLE_E_STATE	operation of the component has not been started

A6.2 CONTROL OF TRACES

Name ISLE_TraceControl
GUID {D020B00B-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_TraceControl.H

The interface is exported by objects that support generation of diagnostic traces. Trace records are entered to the interface ISLE_Trace passed to the method StartTrace(). This interface is provided by the SLE Application. Trace records and the trace levels are specified in 3.6.3.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_Trace;

#define IID_ISLE_TraceControl_DEF { 0xd020b00b, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_TraceControl : IUnknown
{
  virtual HRESULT
    StartTrace( ISLE_Trace* trace,
                SLE_TraceLevel level,
                bool forward ) = 0;
  virtual HRESULT
    StopTrace() = 0;
};
```

Methods

`HRESULT StartTrace( ISLE_Trace* ptrace,`
`SLE_TraceLevel level,`
`bool forward );`

Starts tracing by the object that exports the interface. If the argument forward is set to true, the object also starts tracing of associated lower layers of the API, if applicable.

Arguments

ptrace	pointer to the interface to which trace records shall be passed
level	the trace level that shall be applied as defined in 3.6.3
forward	if set to true, tracing for lower layers of the API shall be started as well

Result codes

S_OK	tracing started
SLE_E_STATE	tracing already active
E_FAIL	the request fails because of a further unspecified error

`HRESULT StopTrace();`

Stops a previously started trace.

Result codes

S_OK	tracing stopped
SLE_E_STATE	tracing not active

A7 SLE API PROXY

A7.1 COMPONENT CREATOR FUNCTION

File `<impl-id>.H`

The API proxy component includes a function to create an instance and obtain a pointer to the administrative interface. The signature of this function is defined as:

```
extern "C" HRESULT
  <impl-id>_CreateProxy( const GUID& iid,
                         void** ppv );
```

where `<impl-id>` is replaced by the product identifier of the implementation. Note that external 'C' linkage is required. The function ensures that a single instance of the proxy is created and returns pointer to the same instance if it is called repetitively. The function checks the argument identifying the interface and returns an error when the implementation does not support an interface with that identifier.

Arguments

`iid`	identifier of the required interface
`ppv`	pointer to the requested interface of the API proxy

Result codes

`S_OK`	the object has been created
`E_NOINTERFACE`	the specified interface is not supported

A7.2 SLE PROXY ADMINISTRATIVE INTERFACE

Name ISLE_ProxyAdmin
GUID {D020B00C-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_ProxyAdmin.H

The interface provides the means to configure the proxy component and to pass it the interfaces needed operationally. All static configuration parameters needed by the proxy are defined in a configuration file. The path name of that file is supplied to the proxy via this interface.

In addition, the interface provides methods to register and de-register ports for a specific service instance. These methods are used by the service element when a service instance is created and deleted. Port registration is described in 3.2.5.

The interface finally provides a method for shutdown of the proxy.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
#include <SLE_Types.h>
interface ISLE_Locator;
interface ISLE_OperationFactory;
interface ISLE_UtilFactory;
interface ISLE_Reporter;
interface ISLE_SII;

#define IID_ISLE_ProxyAdmin_DEF { 0xd020b00c, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_ProxyAdmin : IUnknown
{
  virtual HRESULT
    Configure( const char* configFilePath,
               ISLE_Locator* plocator,
               ISLE_OperationFactory* popFactory,
               ISLE_UtilFactory* putilFactory,
               ISLE_Reporter* preporter ) = 0;
  virtual HRESULT
    ShutDown() = 0;
  virtual HRESULT
    RegisterPort( const ISLE_SII& sii,
                  const char* responderPort,
                  SLE_PortRegId& regId ) = 0;
  virtual HRESULT
    DeregisterPort( SLE_PortRegId regId ) = 0;
  virtual const char*
    Get_ProtocolId() const = 0;
};
```

Methods

```
HRESULT Configure( const char* configFilePath,
                   ISLE_Locator* plocator,
                   ISLE_OperationFactory* popFactory,
                   ISLE_UtilFactory* putilFactory,
                   ISLE_Reporter* preporter );
```

Configures the proxy and passes it the basic interfaces of other components needed for operations. As part of this method, the proxy also configures and initializes the communications system. Any problems and errors are entered into the system log using the interface passed as argument.

Arguments

configFilePath	full path name of the proxy configuration file; the contents of this file is implementation dependent
plocator	Pointer to the locator interface for incoming calls; if no incoming calls are to be accepted, this argument is set to NULL
popFactory	pointer to the operation object factory to be used by the proxy
putilFactory	pointer to the factory interface for utility objects to be used by the proxy
preporter	pointer to the reporter interface for passing of log messages and notifications to the application

Result codes

S_OK	configuration completed without errors
SLE_E_NOFILE	configuration file not found
SLE_E_CONFIG	errors or inconsistencies in the configuration data
SLE_E_COMMS	unable to initialize communications system
E_INVALIDARG	one of the input arguments is NULL
E_FAIL	the request fails because of a further unspecified error

```
HRESULT ShutDown();
```

Requests the proxy to shutdown and release all resources.

Result codes

S_OK	the proxy no longer exists
SLE_E_STATE	operation of the proxy must be terminated first

```
HRESULT RegisterPort( const ISLE_SII& sii,
                      const char* responderPort,
                      SLE_PortRegId& regId );
```

Registers a port. Port registration actions are technology and implementation dependent. The method must be called for every new service instance responding to BIND invocations, as the proxy may depend on this procedure.

Arguments

`sii`	service instance identifier
`responderPort`	logical name of the local port on which the proxy shall accept a BIND invocation
`regId`	registration identifier that must be passed to the proxy when the port is de-registered; for the client the registration identifier is an opaque type and no further meaning should be associated with it; in particular the registration id need not be unique for service instances

Result codes

`S_OK`	port has been registered
`SLE_E_UNKNOWN`	the port identifier is not defined in the configuration database
`SLE_E_INVALIDID`	the port is not defined as a local port
`SLE_E_DUPLICATE`	duplicate registration
`E_NOTIMPL`	the responder role is either not supported or has been disabled by configuration
`E_FAIL`	the request fails because of a further unspecified error

`HRESULT DeregisterPort( SLE_PortRegId regId );`

De-registers a port that has been previously registered.

Arguments

`regId`	registration identifier obtained from a previous call to `RegisterPort()`

Result codes

`S_OK`	port has been de-registered
`SLE_E_UNKNOWN`	port was not registered
`E_NOTIMPL`	the responder role is either not supported or has been disabled by configuration
`E_INVALIDARG`	the registration identifier is invalid

`const char* Get_ProtocolId() const;`

Returns the identifier for the protocol supported by the proxy.

A7.3 ASSOCIATION FACTORY

Name ISLE_AssocFactory
GUID {D020B00D-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_AssocFactory.H

The interface allows creation of associations that take the initiator role for the BIND operation. Associations created via this interface can be used for several consecutive associations for the same service instance. When the association is no longer needed, the proxy must be instructed to destroy the association. In addition, clients must make sure that all references on the interface have been released.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_Types.h>
#include <SLE_APITypes.h>
interface ISLE_SrvProxyInitiate;
interface ISLE_SrvProxyInform;

#define IID_ISLE_AssocFactory_DEF { 0xd020b00d, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_AssocFactory : IUnknown
{
  virtual HRESULT
    CreateAssociation( const GUID& iid,
                       SLE_ApplicationIdentifier srvType,
                       ISLE_SrvProxyInform* pclientIf,
                       void** ppv ) = 0;
  virtual HRESULT
    DestroyAssociation( IUnknown* passoc ) = 0;
};
```

Methods

`HRESULT CreateAssociation( const GUID& iid,`
`SLE_AppplicationIdentifier srvType,`
`ISLE_SrvProxyInform* pclientIf,`
`void** ppv );`

Creates a new association of the specified service type, which acts as an initiator for the BIND operation. If the proxy does not support the service type or the specified interface it returns an error.

Arguments

`iid`	identifier for the interface `ISLE_SrvProxyInitiate`
`srvType`	the SLE service type to be supported by the association
`pclientIf`	pointer to the client interface
`ppv`	pointer to the requested interface of the association

Result codes

`S_OK`	the association object has been created
`SLE_E_STATE`	the proxy has not been started
`E_NOTIMPL`	the service type is not supported by the proxy
`E_NOINTERFACE`	the interface is not supported by an association object

`HRESULT DestroyAssociation( IUnknown* passoc );`

Deletes an association previously created by this interface.

Arguments

`passoc`	pointer to the association object

Result codes

`S_OK`	the object has been destroyed
`SLE_E_STATE`	the association is not in the state unbound
`SLE_E_UNKNOWN`	the association is not known to the proxy
`SLE_E_TYPE`	the association has not been created by this interface

A7.4 SLE SERVICE PROXY INTERFACE

Name ISLE_SrvProxyInitiate
GUID {D020B00E-CCD1-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_SrvProxyInitiate.H

The interface allows a client to pass SLE operation invocations and returns to an association in the proxy for transmission to the peer system.

The association accepts any operation that is valid for the given service type, independent of the service instance state and whether the clients acts as an SLE service user or provider. The only checks applied are related to the state of the association.

For a description of the associated state table of an association see 4.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_Operation;
interface ISLE_ConfirmedOperation;

#define IID_ISLE_SrvProxyInitiate_DEF { 0xd020b00e, 0xccd1, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_SrvProxyInitiate : IUnknown
{
  virtual HRESULT
    InitiateOpInvoke( ISLE_Operation* poperation,
                      bool reportTransmission = false,
                      unsigned long seqCount = 0 ) = 0;
virtual HRESULT
    InitiateOpReturn( ISLE_ConfirmedOperation* poperation,
                      bool report = false,
                      unsigned long seqCount = 0 ) = 0;
virtual HRESULT
    DiscardBuffer() = 0;
  virtual SLE_AssocState
    Get_AssocState() const = 0;
};
```

Methods

`HRESULT InitiateOpInvoke( ISLE_Operation* poperation,`
`bool reportTransmission = false,`
`unsigned long seqCount = 0 );`

Queues the operation invocation defined by the argument `poperation` for transmission. If the argument `reportTransmission` is set to true final transmission is reported via the

interface `ISLE_SrvProxyInform`. If the operation is confirmed, the association returns it when the associated return arrives.

Arguments

`poperation`	the operation object containing the invocation that shall be transmitted
`reportTransmission`	`true` indicates that transmission of the PDU shall be reported
`seqCount`	sequence count for PDUs as defined in 3.7.3

Result codes

`SLE_S_TRANSMITTED`	the PDU has been passed to the communications system for transmission
`SLE_S_QUEUED`	the PDU has been queued locally for transmission
`SLE_E_UNBINDING`	the PDU can no longer be accepted because an UNBIND operation has already been initialized
`SLE_E_INVALIDID`	the identifier of the peer application passed in a BIND invocation is not defined in the configuration database
`SLE_E_INVALIDPDU`	the operation is not supported for the service type
`SLE_E_PROTOCOL`	the operation cannot be accepted in the current state, because that would result in a protocol error
`SLE_E_COMMS`	the request cannot be performed because of a communications system failure
`SLE_E_OVERFLOW`	the configured queuing capability has been exceeded
`SLE_E_ABORTED`	the association has been aborted
`SLE_E_SEQUENCE`	sequence count out of acceptable window
`E_FAIL`	the request fails because of a further unspecified error

```
HRESULT InitiateOpReturn( ISLE_ConfirmedOperation* poperation,
                          bool reportTransmission = false,
                          unsigned long seqCount = 0 );
```

Queues the operation return defined by the argument `poperation` for transmission. If the argument `reportTransmission` is set to true final transmission is reported via the interface `ISLE_SrvProxyInform`.

Arguments

`poperation`	the operation object containing the invocation that shall be transmitted
`reportTransmission`	`true` indicates that transmission of the PDU shall be reported
`seqCount`	sequence count for PDUs as defined in 3.7.3

Result codes

SLE_S_TRANSMITTED	the PDU has been passed to the communications system for transmission
SLE_S_QUEUED	the PDU has been queued locally for transmission
SLE_E_UNBINDING	the PDU can no longer be accepted because an UNBIND operation has already been initialized
SLE_E_INVALIDPDU	the operation is not supported for the service type
SLE_E_PROTOCOL	the operation cannot be accepted in the current state, because that would result in a protocol error
SLE_E_COMMS	the request cannot be performed because of a communications system failure
SLE_E_OVERFLOW	the configured queuing capability has been exceeded
SLE_E_ABORTED	the association has been aborted
SLE_E_SEQUENCE	sequence count out of acceptable window
E_FAIL	the request fails because of a further unspecified error

`HRESULT DiscardBuffer();`

Searches the local transmission queue of operations of the type TRANSFER-BUFFER, and deletes all objects for which transmission of data has not yet started. Returns whether any buffer has been discarded.

Result codes

SLE_S_NOTDISCARDED	no buffer deleted
SLE_S_DISCARDED	at least one buffer discarded
SLE_E_STATE	the request is not valid in the current state of the association

`SLE_AssocState Get_AssocState() const;`

Returns the current state of the association.

A8 SLE API SERVICE ELEMENT

A8.1 COMPONENT CREATOR FUNCTION

The API Service Element component includes a function to create an instance and obtain a pointer to the administrative interface. The signature of this function is defined as:

```
extern "C" HRESULT
  <impl-id>_CreateServiceElement( const GUID& iid,
                                  void** ppv );
```

where <impl-id> is replaced by the product identifier of the implementation. Note that external 'C' linkage is required. The function ensures that a single instance of the proxy is created and returns the same instance if it is called repetitively. The function checks the argument identifying the interface and returns an error when the implementation does not support an interface with this identifier.

Arguments

iid	identifier of the required interface
ppv	pointer to the requested interface of the API Service Element

Result codes

S_OK	the object has been created
E_NOINTERFACE	the specified interface is not supported

A8.2 API SERVICE ELEMENT ADMINISTRATIVE INTERFACE

Name ISLE_SEAdmin
GUID {24396FC0-CD99-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_SEAdmin.H

The interface provides the means to configure the service element component and to pass it the interfaces needed operationally. All static configuration parameters needed by the component are defined in a configuration file. The path name of that file is supplied to the proxy via this interface.

Clients must first call the method `Configure()` and then call `AddProxy()` to pass a pointer to the proxy component for every proxy that shall be supported.

The interface finally provides a method for shutdown of the service element.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_OperationFactory;
interface ISLE_UtilFactory;
interface ISLE_Reporter;
interface ISLE_ProxyAdmin;

#define IID_ISLE_SEAdmin_DEF { 0x24396fc0, 0xcd99, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_SEAdmin : IUnknown
{
  virtual HRESULT
    Configure( const char* configFilePath,
               ISLE_OperationFactory* popFactory,
               ISLE_UtilFactory* putilFactory,
               ISLE_Reporter* preporter ) = 0;
virtual HRESULT
    AddProxy( const char* protocolId,
              SLE_BindRole role,
              ISLE_ProxyAdmin* pproxy ) = 0;
  virtual HRESULT
    ShutDown() = 0;
};
```

Methods

```
HRESULT Configure( const char* configFilePath,
                   ISLE_OperationFactory* popFactory,
                   ISLE_UtilFactory* putilFactory,
                   ISLE_Reporter* preporter );
```

Configures the service element component and passes it the basic interfaces of other components needed for operations. Any problems and errors are entered into the system log using the interface passed as argument.

Arguments

`configFilePath`	full path name of the configuration file; the contents of this file is implementation dependent
`popFactory`	pointer to the operation object factory to be used by the service element
`putilFactory`	pointer to the factory interface for utility objects to be used by the service element
`preporter`	pointer to the reporter interface for passing of log messages and notifications to the application

Result codes

`S_OK`	configuration completed without errors
`SLE_E_NOFILE`	configuration file not found
`SLE_E_CONFIG`	errors or inconsistencies in the configuration data
`E_INVALIDARG`	one of the input arguments is NULL
`E_FAIL`	the request fails because of a further unspecified error

```
HRESULT AddProxy( const char* protocolId,
                  SLE_BindRole role,
                  ISLE_ProxyAdmin* pproxy );
```

Passes the proxy component to use to the service element.

Arguments

`protocolId`	identification of the technology and mapping supported by the proxy; this argument is required for selection of the correct proxy when multiple proxies are configured
`role`	the bind roles supported by the API proxy component
`pproxy`	pointer to the administrative interface of the proxy

Result codes

`S_OK`	proxy added
`SLE_E_OVERFLOW`	too many proxies
`SLE_E_DUPLICATE`	protocol identifier already used by a configured proxy
`E_FAIL`	the request fails because of a further unspecified error

A8.3 SERVICE INSTANCE LOCATOR

Name ISLE_Locator
GUID {24396FC1-CD99-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_Locator.H

The interface is provided to the proxy to obtain an interface of the type ISLE_SrvProxyInform when a BIND invocation has been received. When an error is returned, the proxy is expected to reject the BIND invocation.

Synopsis

```
#include <SLE_SCM.H>

interface ISLE_Bind;
interface ISLE_SrvProxyInform;
interface ISLE_SrvProxyInitiate;

#define IID_ISLE_Locator_DEF { 0x24396fc1, 0xcd99, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Locator : IUnknown
{
  virtual HRESULT
    LocateInstance( ISLE_SrvProxyInitiate* passociation,
                    ISLE_Bind* pbindop,
                    ISLE_SrvProxyInform** ppServiceInstance) = 0;
};
```

Methods

```
HRESULT LocateInstance( ISLE_SrvProxyInitiate* passociation,
                        ISLE_Bind* pbindop
                        ISLE_SrvProxyInform** ppServiceInstance);
```

Obtains and returns an interface ISLE_SrvProxyInform for use by a new association. To locate (or create) the object implementing ISLE_SrvProxyInform, the BIND operation is made available, which contains all information needed. The interface ISLE_SrvProxyInitiate is made available to the object providing the returned interface ISLE_SrvProxyInform.

If no interface can be made available returns an error and sets the output argument to NULL. In this case the proxy is expected to reject the BIND invocation by a BIND return with a negative response and a diagnostic corresponding to the returned error.

An implementation is not required to perform all the checks defined by the result codes in this method. It can also accept the association and perform the checks when the BIND invocation is passed to the interface ISLE_SrvProxyInform.

Arguments

`passociation`	interface provided by the association on which the BIND invocation was received
`pbindop`	bind operation object holding the received bind invocation
`ppServiceInstance`	complementary interface that shall be used by the association to forward PDUs received from the network

Result codes

`S_OK`	a service instance has been located and is ready to accept the BIND invocation
`SLE_E_UNKNOWN`	the service instance identifier in the BIND invocation does not match any available service instance
`E_ACCESSDENIED`	the service instance does not belong to the peer application as identified by the application identifier in the BIND operation
`SLE_E_TYPE`	the service type specification in the BIND operation does not match the service type in the service instance
`SLE_E_TIME`	the scheduled provision period of the service instance has not yet started or has expired
`SLE_E_STATE`	the service instance is already bound
`E_FAIL`	the request fails because of a further unspecified error

A8.4 SLE SERVICE INSTANCE FACTORY

Name ISLE_SIFactory
GUID {BB4DDA2E-54CD-11d8-9CF5-0004761E8CFB}
Inheritance: IUnknown
File ISLE_SIFactory.H

The interface allows creation of service instances for a specified service type and for a specified role (SLE service provider or SLE service user). Following creation, the service instance must be configured using its administrative interface. When the association is no longer needed, the service element component must be instructed to destroy the service instance. In addition, clients must make sure that all references on all interfaces of the service instance have been released.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
#include <SLE_Types.h>
interface ISLE_ServiceInform;

#define IID_ISLE_SIFactory_DEF { 0xbb4dda2e, 0x54cd, 0x11d8, \
          { 0x9c, 0xf5, 0x0, 0x4, 0x76, 0x1e, 0x8c, 0xfb } }

interface ISLE_SIFactory : IUnknown
{
  virtual HRESULT
    CreateServiceInstance( const GUID& iid,
                           SLE_ApplicationIdentifier srvType,
                           SLE_VersionNumber version,
                           SLE_AppRole role,
                           ISLE_ServiceInform* pclientIf,
                           void** ppv ) = 0;
  virtual HRESULT
    DestroyServiceInstance( IUnknown* psi ) = 0;
};
```

Methods

`HRESULT CreateServiceInstance( const GUID& iid,`
`SLE_ApplicationIdentifier srvType,`
`SLE_VersionNumber version,`
`SLE_AppRole role,`
`ISLE_ServiceInform* pclientIf,`
`void** ppv );`

Creates a new service instance for the requested SLE service type, supporting the requested role (SLE service provider or SLE service user). Returns a pointer to the requested interface of the service instance. If the component does not support the service type, the requested role, or the requested interface identifier returns an error.

For service instances in the user (initiator) role the version of the SLE service must be specified and this specification will be included into the BIND invocation. On the provider (responder) side the version number is defined by the BIND invocation received from the user and checked against the list of supported version numbers in the configuration database of the API Proxy. Therefore the argument is ignored and should be set to zero.

Arguments

`iid`	the identifier for the required interface
`srvType`	the SLE service type to be supported by the service instance
`version`	for the user (initiator) role defines the version number of the SLE service type to be used; for the provider (responder) side this argument is ignored and should be set to zero
`role`	the role (user or provider) to be supported by the service instance
`pclientIf`	pointer to the interface the service instance shall use to pass operations to the client
`ppv`	pointer to the requested interface of the service instance

Result codes

`S_OK`	the service instance object has been created
`SLE_E_STATE`	the service element has not been started
`SLE_E_INVALIDID`	the version number is zero for the role ‘user’
`E_NOTIMPL`	the service type or the version or the role is not supported by the service element
`E_NOINTERFACE`	the interface is not supported by an association object

`HRESULT DestroyServiceInstance( IUnknown* psi );`

Destroys a service instance created by this interface.

Arguments

`psi`	pointer to the service interface

Result codes

`S_OK`	service instance destroyed
`SLE_E_UNKNOWN`	the service instance is not known
`SLE_E_STATE`	the service instance is not in the unbound state; the association must be aborted first

A8.5 SLE SERVICE INSTANCE ADMINISTRATIVE INTERFACE

Name `ISLE_SIAdmin`
GUID `{BB4DDA31-54CD-11d8-9CF5-0004761E8CFB}`
Inheritance: `IUnknown`
File `ISLE_SIAdmin.H`

The interface is provided for configuration of service instances. It can be used for instances supporting the provider role or the user role. For instances supporting the user role not all parameters need to be set.

Clients must specify the individual parameters using the method foreseen for the parameter. Depending on the service type, further parameters may have to be supplied using the service type-specific configuration interface. When all parameters have been supplied, the method `ConfigCompleted()` must be called. The service instance then verifies that the configuration is complete and consistent and performes all actions required to start nominal operation. If the method `ConfigCompleted()` returns with success, the service instance is ready for operation.

As a general precondition, configuration parameters must not be modified after a successful return of the method `ConfigCompleted()`. The effect of an attempt to set a parameter when the initial configuration has completed, is undefined.

The interface provides read access to all configuration parameters, including those defined in the create request to the Service Instance Factory. The value returned by a call to the read methods before configuration has been completed, is generally undefined.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
#include <SLE_Types.h>
interface ISLE_SII;
interface ISLE_Time;

#define IID_ISLE_SIAdmin_DEF { 0xbb4dda31, 0x54cd, 0x11d8, \
          { 0x9c, 0xf5, 0x0, 0x4, 0x76, 0x1e, 0x8c, 0xfb } }


interface ISLE_SIAdmin : IUnknown
{
  virtual void
    Set_ServiceInstanceId( const ISLE_SII& id ) = 0;
  virtual void
    Put_ServiceInstanceId( ISLE_SII* id ) = 0;
  virtual void
    Set_PeerIdentifier( const char* id ) = 0;
  virtual void
    Set_ProvisionPeriod( const ISLE_Time* start,
                         const ISLE_Time* stop ) = 0;
  virtual void
```

```
    Set_BindInitiative( SLE_AppRole role ) = 0;
  virtual void
    Set_ResponderPortIdentifier( const char* portId ) = 0;
  virtual void
    Set_ReturnTimeout( int timeout ) = 0;
  virtual HRESULT
    ConfigCompleted() = 0;
  virtual SLE_ApplicationIdentifier
    Get_ServiceType() const = 0;
  virtual SLE_VersionNumber
    Get_Version() const = 0
  virtual SLE_AppRole
    Get_Role() const = 0;
  virtual const ISLE_SII*
    Get_ServiceInstanceIdentifier() const = 0;
  virtual const char*
    Get_PeerIdentifier() const = 0;
  virtual const ISLE_Time*
    Get_ProvisionPeriodStart() const = 0;
  virtual const ISLE_Time*
    Get_ProvisionPeriodStop() const = 0;
  virtual SLE_AppRole
    Get_BindInitiative() const = 0;
  virtual const char*
    Get_ResponderPortIdentifier() const = 0;
  virtual int
    Get_ReturnTimeout() const = 0;
};
```

Methods

`void Set_ServiceInstanceId( const ISLE_SII& id );`

Sets the service instance identifier copying the input argument.

`void Put_ServiceInstanceId( ISLE_SII* id );`

Sets the service instance to the input argument. The argument will be deleted by the service instance object.

`void Set_PeerIdentifier( const char* id );`

Sets the identifier of the peer application.

```
void Set_ProvisionPeriod( const ISLE_Time* start,
                          const ISLE_Time* stop );
```

Sets the scheduled provisioning period according the start and stop times passed as arguments. If the start time is NULL, the service instance assumes immediate start of the provision period. If the stop time is NULL, the service instance provision period never expires.

`void Set_BindInitiative( SLE_AppRole role );`

Specifies whether user-initiated binding or server-initiated binding shall be used.

`void Set_ResponderPortIdentifier( const char* portId );`

Sets the port identifier for the responding application.

`void Set_ReturnTimeout( int timeout );`

Sets the timeout value in which a return for confirmed operations must arrive. The timeout argument is passed in units of seconds.

`HRESULT ConfigCompleted();`

Checks the configuration of the service element on completeness and consistency and performs all actions needed to start nominal operation. The method includes checking of all service type-specific parameters and takes into account the role (user or provider) of the service instance. This method must not be called again after successful completion indicated by a result code of `S_OK`.

Result codes

`S_OK`	all checks passed; the service instance is ready for operation
`E_NOTIMPL`	provider-initiated bind not supported
`SLE_E_TIME`	inconsistent start and stop times
`SLE_E_PORT`	the port identifier does not match the configuration or could not be registered
`SLE_E_INVALIDID`	invalid service instance identifier
`SLE_E_STATE`	the service instance is already configured
`SLE_E_CONFIG`	other, further unspecified configuration problem

`SLE_ApplicationIdentifier Get_ServiceType() const;`

Returns the service type supported by the service instance.

`SLE_VersionNumber Get_Version() const;`

Returns the version number of the service type supported by the service instance. For the role ‘provider’ returns the value extracted from the received BIND invocation when the service instance is bound and zero when the service instance is not bound.

`SLE_AppRole Get_Role() const;`

Returns the application role (user or provider) assumed by the service instance.

`const SLE_SII* Get_ServiceInstanceIdentifier() const;`

Returns the service instance identifier set in the object, or NULL if no identifier has been set.

`const char* Get_PeerIdentifier() const;`

Returns the peer identifier set in the service instance or NULL when not yet configured.

`const ISLE_Time* Get_ProvisionPeriodStart() const;`

Returns the provisioning start time set in the service instance or NULL when not yet configured.

`const ISLE_Time* Get_ProvisionPeriodStop() const;`

Returns the provisioning stop time set in the service instance or NULL when not yet configured.

`SLE_AppRole Get_BindInitiative() const;`

Returns the bind initiative (user-initiated or provider-initiated) set for the service instance.

`const char* Get_ResponderPortIdentifier() const;`

Returns the logical port identifier set in the service instance or NULL when not yet configured.

`int Get_ReturnTimeout() const;`

Returns the return timeout period set in the service instance or 0 when not yet configured.

A8.6 SLE SERVICE INTERFACES

A8.6.1 Service Proxy Interface

Name `ISLE_SrvProxyInform`
GUID `{24396FC4-CD99-11d2-9B44-00A0246D80DB}`
Inheritance: `IUnknown`
File `ISLE_SrvProxyInform.H`

The interface is provided to the proxy for transfer of operation invocations and returns on a single association. In addition, it provides a method to signal transfer of a PDU if that has been requested via the complementary interface `ISLE_SrvProxyInitiate`.

The PDUs passed via this interface are generally unchecked. The only checks performed by the proxy are that the PDU is supported by the service type and is properly coded. Reception of an invalid PDU via this interface shall not cause the function to be rejected. The provider of the interface must either generate the appropriate operation return or abort the association.

Calls to this interface shall only be rejected when the client misbehaves. For instance, passing of an invocation other than BIND in the state unbound is such an error.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_Operation;
interface ISLE_ConfirmedOperation;

#define IID_ISLE_SrvProxyInform_DEF { 0x24396fc4, 0xcd99, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_SrvProxyInform : IUnknown
{
  virtual HRESULT
    InformOpInvoke( ISLE_Operation* poperation,
                    unsigned long seqCount = 0 ) = 0;
  virtual HRESULT
    InformOpReturn( ISLE_ConfirmedOperation* poperation,
                    unsigned long seqCount = 0 ) = 0;
  virtual HRESULT
    PDUTransmitted( ISLE_Operation* poperation ) = 0;
  virtual HRESULT
    ProtocolAbort( const SLE_Octet* diagnostic,
                   size_t size ) = 0;
};
```

Methods

```
HRESULT InformOpInvoke( ISLE_Operation* poperation,
                        unsigned long seqCount = 0 );
```

Accepts an operation invocation.

Preconditions: The invocation was received on the association and was correctly decoded

Arguments

`poperation`	the operation object containing the invocation
`seqCount`	sequence count for PDUs as defined in 3.7.3

Result codes

`S_OK`	PDU accepted
`SLE_E_PROTOCOL`	the request violates the state machine and should have been prevented by the proxy
`SLE_E_INVALIDPDU`	the PDU is not valid for the service type
`SLE_E_SEQUENCE`	sequence count out of acceptable window

```
HRESULT InformOpReturn( ISLE_ConfirmedOperation* poperation,
                        unsigned long seqCount = 0 );
```

Accepts an operation return.

Preconditions: The return was received on the association and was correctly decoded

Arguments

`poperation`	the operation object containing the return
`seqCount`	sequence count for PDUs as defined in 3.7.3

Result codes

`S_OK`	PDU accepted
`SLE_E_PROTOCOL`	the request violates the state machine and should have been prevented by the proxy
`SLE_E_INVALIDPDU`	the PDU is not valid for the service type
`SLE_E_UNSOLICITED`	the operation was not previously passed to the association by this service instance
`SLE_E_SEQUENCE`	sequence count out of acceptable window

```
HRESULT PDUTransmitted( ISLE_Operation* poperation );
```

Reports transmission of a PDU as requested via the interface `ISLE_SrvProxyInitiate`.

Arguments

`poperation`	pointer to the operation for which a report was requested; the operation object may no longer exists and the pointer should not be used for access to the operation object unless the receiver still holds a reference to the operation object

Result codes

`S_OK`	report accepted
`SLE_E_UNSOLICITED`	no report had been requested

`HRESULT ProtocolAbort( const SLE_Octet* diagnostic, size_t size );`

Reports failure of the communications system for the association.

Arguments

`diagnostic`	diagnostic data as provided by the data communication service
`size`	size of the diagnostic data in bytes

Result codes

`S_OK`	accepted
`E_UNEXPECTED`	the service instance is not aware of an active association

A8.6.2 Service Application Interface

Name `ISLE_ServiceInitiate`
GUID `{7B425727-D32D-11d2-9B44-00A0246D80DB}`
Inheritance: `IUnknown`
File `ISLE_ServiceInitiate.H`

The interface provides the methods to pass operation invocations and returns to a service instance in the API Service Element. The requests are checked and an error is returned in case of incomplete or inconsistent definitions or if the PDU is not valid in the current state of the service instance. For the definition of the state table see section 4. A positive return code of the methods ensures that the PDU has been queued for transmission. It does not indicate that the PDU has been actually transmitted.

For the following special PDUs, the interface may return the code `SLE_S_SUSPEND` indicating that further transfer of data shall be suspended:

a) TRANSFER DATA Invocation for forward Services;

b) TRANSFER DATA Invocation for Return Services when the delivery mode is either complete online or offline.

The method `ResumeDataTransfer()` will be called on the complementary interface `ISLE_ServiceInform` when data transfer is again possible.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_Operation;
interface ISLE_ConfirmedOperation;

#define IID_ISLE_ServiceInitiate_DEF { 0x7b425727, 0xd32d, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_ServiceInitiate : IUnknown
{
  virtual HRESULT
    InitiateOpInvoke( ISLE_Operation* poperation,
                      unsigned long seqCount = 0 ) = 0;
  virtual HRESULT
    InitiateOpReturn( ISLE_ConfirmedOperation* poperation,
                      unsigned long seqCount = 0 ) = 0;
  virtual SLE_SIState
    Get_SIState() const = 0;
};
```

Methods

```
HRESULT InitiateOpInvoke( ISLE_Operation* poperation,
                          unsigned long seqCount = 0 );
```

Accepts an operation invocation for transfer. If the operation is confirmed, the operation object will be passed back via the complementary interface when the return is received.

Arguments

poperation	the operation object containing the invocation
seqCount	sequence count for PDUs as defined in 3.7.3

Result codes

S_OK	PDU queued for transmission
SLE_S_SUSPEND	PDU queued but now suspend data transfer
SLE_E_PROTOCOL	the PDU cannot be accepted in the current state because that would result in a protocol error protocol error
SLE_E_MISSINGARG	at least one argument is missing in the PDU
SLE_E_INCONSISTENT	the arguments in the PDU are inconsistent
SLE_E_RANGE	at least one argument in the PDU is out of range
SLE_E_INVALIDID	the identifier of the peer application passed in a BIND invocation is not defined in the configuration database
SLE_E_INVALIDPDU	the PDU is not valid for the service type
SLE_E_ROLE	the PDU is not valid for the role of the service instance
SLE_E_SUSPENDED	the PDU cannot be accepted because data transfer is currently suspended
SLE_E_UNBINDING	the PDU can no longer be accepted because an UNBIND operation has already been invoked
SLE_E_STOPPING	the PDU can no longer be accepted because a STOP operation has already been invoked
SLE_E_ABORTED	the association has been aborted
SLE_E_OVERFLOW	the configured queuing capability has been exceeded; the association has been aborted
SLE_E_SEQUENCE	sequence count out of acceptable window
SLE_E_COMMS	the request cannot be performed because of a communications system failure

```
HRESULT InitiateOpReturn( ISLE_ConfirmedOperation* poperation,
                          unsigned long seqCount = 0 );
```

Accepts an operation return for transfer. The operation object must have been passed to the application via the complementary interface.

Arguments

poperation	the operation object containing the return
seqCount	sequence count for PDUs as defined in 3.7.3

Result codes

S_OK	PDU queued for transmission
SLE_E_PROTOCOL	the PDU cannot be accepted in the current state because that would result in a protocol error protocol error
SLE_E_MISSINGARG	at least one argument is missing in the PDU
SLE_E_INCONSISTENT	the arguments in the PDU are inconsistent
SLE_E_RANGE	at least one argument in the PDU is out of range
SLE_E_INVALIDPDU	the PDU is not valid for the service type
SLE_E_ROLE	the PDU is not valid for the role of the service instance
SLE_E_UNSOLICITED	the operation was not previously passed to the application by this service instance
SLE_E_UNBINDING	the PDU can no longer be accepted because an UNBIND operation has already been invoked
SLE_E_STOPPING	the PDU can no longer be accepted because a STOP operation has already been invoked
SLE_E_ABORTED	the association has been aborted
SLE_E_OVERFLOW	the configured queuing capability has been exceeded; the association has been aborted
SLE_E_SEQUENCE	sequence count out of acceptable window
SLE_E_COMMS	the request cannot be performed because of a communications failure

`SLE_SIState Get_SIState();`

Returns the current state of the service instance.

A8.7 SERVICE SPECIFIC OPERATION FACTORY

Name ISLE_SIOpFactory
GUID {24396FC5-CD99-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_SIOpFactory.H

The interface defines a specialized operation object factory provided by service instances in the API Service Element. The interface is able to create operation objects for a given service type and service role. Operation objects created via this interface are 'pre-configured' using the data specified for the service instance (see the description of the operation object interfaces for details). Operation objects for common association management (see A5.4) are created by all service instances.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_Operation;

#define IID_ISLE_SIOpFactory_DEF { 0x24396fc5, 0xcd99, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_SIOpFactory : IUnknown
{
  virtual HRESULT
    CreateOperation( const GUID& iid,
                     SLE_OpType optype,
                     void** ppv ) const = 0;
};
```

Methods

```
HRESULT CreateOperation( const GUID& iid,
                         SLE_OpType optype,
                         void** ppv );
```

Creates and configures a new operation object as specified by the argument. If the interface cannot be found, does not refer to an operation object interface, or is not supported for the service type or role, returns an error.

Arguments

iid	GUID for the interface to be returned
optype	type of the operation object to be created
ppv	pointer to the requested interface of the operation object

Result codes

S_OK	the operation object has been created and configured
SLE_E_STATE	the service instance has not been configured
SLE_E_TYPE	the operation is not supported by the service type or for the role of the service instance (e.g., a RAF Service User application cannot create a TRANSFER-DATA operation)
E_NOINTERFACE	the interface is not supported by an operation object of the specified type

A9 SLE API APPLICATION INTERFACES

A9.1 SLE SERVICE INTERFACE

Name ISLE_ServiceInform
GUID {24396FC6-CD99-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_ServiceInform.H

The interface allows passing of operation invocations and returns to the SLE Application for a single service instance. The client of this interface is expected to have checked all PDUs that pass this interface to the level defined for the API Service Element in 3.3.5.1.2.

For data transfer for forward services and for the complete online and offline return services, the interface additionally provides the means to signal to the application that data transfer may continue. Suspension of data transfer is requested via the complementary interface ISLE_ServiceInitiate.

Finally the interface provides a means to inform the application when a protocol abort occurs and when the scheduled provision period of the service instance ends.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_Operation;
interface ISLE_ConfirmedOperation;

#define IID_ISLE_ServiceInform_DEF { 0x24396fc6, 0xcd99, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_ServiceInform : IUnknown
{
  virtual HRESULT
    InformOpInvoke( ISLE_Operation* poperation,
                    unsigned long seqCount = 0 ) = 0;
  virtual HRESULT
    InformOpReturn( ISLE_ConfirmedOperation* poperation,
                    unsigned long seqCount = 0 ) = 0;
  virtual void
    ResumeDataTransfer() = 0;
  virtual void
    ProvisionPeriodEnds() = 0;
  virtual HRESULT
    ProtocolAbort( const SLE_Octet* diagnostic,
                   size_t size ) = 0;
};
```

Methods

```
HRESULT InformOpInvoke( ISLE_Operation* poperation,
                        unsigned long seqCount = 0 );
```

Accepts an operation invocation. If the operation is confirmed, it shall be passed back to the service instance via the complementary interface when the return is sent.

Arguments

`poperation`	the operation object containing the invocation
`seqCount`	sequence count for PDUs as defined in 3.7.3

Result codes

`S_OK`	the invocation has been accepted
`SLE_E_SEQUENCE`	sequence count out of acceptable window
`E_FAIL`	further unspecified error

```
HRESULT InformOpReturn( ISLE_ConfirmedOperation* poperation,
                        unsigned long seqCount = 0 );
```

Accepts an operation return.

Arguments

`poperation`	the operation object containing the return
`seqCount`	sequence count for PDUs as defined in 3.7.3

Result codes

`S_OK`	the invocation has been accepted
`SLE_E_SEQUENCE`	sequence count out of acceptable window
`E_FAIL`	further unspecified error

```
void ResumeDataTransfer();
```

Informs the application that a previously suspended data transfer activity can be resumed.

```
void ProvisionPeriodEnds();
```

Informs the application that the scheduled provision period for the service instance ends and the service instance can be deleted.

```
HRESULT ProtocolAbort( const SLE_Octet* diagnostic, size_t size );
```

Reports failure of the communications system for the association.

Arguments

`diagnostic`	diagnostic data as provided by the data communication service
`size`	size of the diagnostic data in bytes

Result codes

S_OK	accepted
E_UNEXPECTED	the application is not aware of an active association

A9.2 REPORTING INTERFACE

Name ISLE_Reporter
GUID {24396FC7-CD99-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_Reported.H

The interface is passed to all API components and is used to enter messages into the system log and to notify the application of specific alarms. The types of alarms, which can be passed to this interface, are defined in 3.6.2.3.1. An alarm is complemented by a brief 20-character text that can be used for display.

The methods in this interface do not report the time of an event. It is expected that the time is added by the implementation of the interface.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_SII;

#define IID_ISLE_Reporter_DEF { 0x24396fc7, 0xcd99, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Reporter : IUnknown
{
  virtual void
    LogRecord( SLE_Component component,
               const ISLE_SII* sii,
               SLE_LogMessageType type,
               unsigned long messageId,
               const char* message ) = 0;
  virtual void
    Notify( SLE_Alarm alarm,
            SLE_Component component,
            const ISLE_SII* sii,
            unsigned long messageId,
            const char* message = 0 ) = 0;
};
```

Methods

```
void LogRecord( SLE_Component component,
                const ISLE_SII* sii,
                SLE_LogMessageType type,
                unsigned long messageId,
                const char* message );
```

Enters a message into the system log.

Arguments

`component`	the component that issues the log message
`sii`	the service instance identifier associated with the log message or a NULL pointer if the message is not associated with a particular service instance
`type`	the type of the log message (alarm or information message)
`messageId`	the numeric identifier of the message
`message`	an ASCII string without formatting characters

```
void Notify( SLE_Alarm alarm,
             SLE_Component component,
             const ISLE_SII* sii,
             unsigned long messageId,
             const char* message );
```

Notifies the application of a specific event.

Arguments

`alarm`	the alarm that is notified
`component`	the components generating the notification
`sii`	the service instance identifier associated with the notification or a NULL pointer if the notification is not associated with a particular service instance
`messageId`	the numeric identifier of the message
`text`	an optional ASCII string of max 20 characters without formatting characters; if no text is supplied the argument must be set to NULL

A9.3 TRACING INTERFACE

Name ISLE_Trace
GUID {24396FC8-CD99-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File ISLE_Trace.H

The interface is provided to API components to enter trace records, when tracing is started via the interface ISLE_TraceControl.

The trace method in this interface does not report the time of an event. It is expected that the time is added by the implementation of the interface.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
interface ISLE_SII;

#define IID_ISLE_Trace_DEF { 0x24396fc8, 0xcd99, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface ISLE_Trace : IUnknown
{
  virtual void
    TraceRecord( SLE_TraceLevel level,
                 SLE_Component component,
                 ISLE_SII* psii,
                 const char* text ) = 0;
};
```

Methods

```
void TraceRecord( SLE_TraceLevel level,
                  SLE_Component component,
                  ISLE_SII* psii,
                  const char* text );
```

Enters a trace record.

Arguments

level	the trace level at which the record is generated
component	the component that issues the trace record
psii	the service instance identifier associated with the trace record or a NULL pointer if the record is not associated with a particular service instance
text	an ASCII string with the trace information

A9.4 TIME SOURCE INTERFACE

Name `ISLE_TimeSource`
GUID `{390d3508-6c7c-11d3-a297-80954a16aa77}`
Inheritance: `IUnknown`
File `ISLE_TimeSource.H`

The interface is provided to the API component 'SLE Utilities' by applications wishing to provide a time source to the API. If provided, the component obtains current time from this interface. Other API components will obtain this time via the utility interface `ISLE_Time`.

For the purpose of implementing timers, API components can assume that the time provided by this interface has a constant offset to system time within the limits of the timer accuracy required by this specification.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

#define IID_ISLE_TimeSource_DEF { 0x390d3508, 0x6c7c, 0x11d3, \
          { 0xa2, 0x97, 0x80, 0x95, 0x4a, 0x16, 0xaa, 0x77 } }

interface ISLE_TimeSource : IUnknown
{
  virtual SLE_Octet*
    Get_CurrentTime() const = 0;
};
```

Methods

`SLE_Octet* Get_CurrentTime() const;`

Returns the current time in CCSDS CDS format.

ANNEX B

RESULT CODES

(Normative)

B1 GENERAL

For the result codes returned by methods in SLE API interfaces this Recommended Practice adopts the scheme defined by the COM Specification (see annex D).

The general COM result code `HRESULT` is a 32 bit integer, structured as shown in figure B-1.

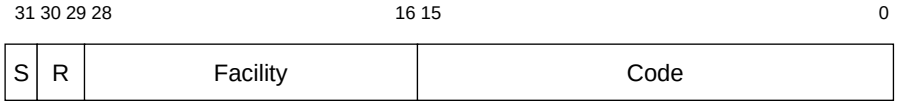

Figure B-1: Structure of Result Codes

The fields in this structure are:

S	Severity field (1 bit) 0 Success 1 Error
R	Reserved (2 bits, must be set to zero)
Facility	indicates a group of codes
Code	the individual code within a facility

COM specifies a number of general codes in the COM Facility NULL (facility code 0), of which a subset is adopted for the SLE API. In order to ensure, that the SLE API can also be implemented in a COM environment, specific codes defined by this Recommended Practice are allocated to the COM Facility ITF (facility code 4), which is foreseen for interface specific result codes. In addition, all specific codes are located above HEX 0200 to avoid conflicts with codes allocated by COM.

Following COM conventions, the mnemonics for the result codes are defined as

`<Facility>_<Severity>_<Reason>`

Where `<Facility>` is always either empty (Facility NULL) or '`SLE`', `<Severity>` is either '`S`' for success or '`E`' for error and `<Reason>` a mnemonic for the reason.

NOTE – If `<Facility>` is empty, then no leading underscore ('_') character will be used in front of `<Severity>`.

The result codes adopted from the COM Facility NULL and the result codes defined by this Recommended Practice are available in the file `SLE_Result.H`. This file also defines the following useful macros also adopted from COM:

`SUCCEEDED(result)`	returns true if the severity is 'success'
`FAILED(result)`	returns true if the severity is 'error'
`HRESULT_CODE(result)`	returns the value of the code field
`HRESULT_FACILITY(result)`	returns the value of the facility field
`HRESULT_SEVERITY(result)`	returns the value of the severity field
`MAKE_HRESULT(sev,fac,code)`	constructs a `HRESULT` variable

The result codes returned by functions of the SLE API are defined in annex A of this Recommended Practice and it is expected that implementations do not return any other code than specified there, except for the following, that can always be returned by a function:

`E_FAIL`	indicating an unspecified error
`E_UNEXPECTED`	indicating a catastrophic failure, which is expected to originate from a serious software problem

In spite of this general rule, experience shows that not all cases can be tested and clients should expect functions to return other codes and process them as a general error indication.

This Recommended Practice reserves the codes HEX 0200 through HEX 03FF for use by the SLE API Recommended Standard and its supplemental Recommended Practice documents for service-specific APIs. It is recommended that implementations of this Recommended Practice allocate additional result codes in the range HEX 0400 through HEX 05FF, leaving the codes HEX 0600 and above for use by SLE applications.

B2 SUCCESS CODES ADOPTED FROM THE COM FACILITY NULL

Mnemonic	Description	Code

`S_OK`
`NOERROR`
`NO_ERROR`
	indicates success	0000

`S_FALSE`	success, but the result is `FALSE`	0001

B3 ERROR CODES ADOPTED FROM THE COM FACILITY NULL

Mnemonic	Description	Code
E_UNEXPECTED	catastrophic failure	FFFF
E_NOTIMPL	not implemented	0001
E_OUTOFMEMORY	ran out of memory	0002
E_INVALIDARG	one or more arguments are invalid	0003
E_NOINTERFACE	the interface is not supported	0004
E_POINTER	invalid pointer	0005
E_ABORT	operation aborted	0007
E_FAIL	unspecified error	0008
E_ACCESSDENIED	general access denied error	0009
E_PENDING	the data necessary to complete this operation are not yet available	000A

B4 SUCCESS CODES DEFINED BY THIS RECOMMENDED PRACTICE (FACILITY ITF)

Mnemonic	Description	Code
SLE_S_TRANSMITTED	the PDU has been passed to the data communications system for transmission	0200
SLE_S_QUEUED	the PDU has been queued for transmission	0201
SLE_S_SUSPEND	suspend data transfer from now on until further notice	0202
SLE_S_DISCARDED	at least one buffer has been discarded	0203
SLE_S_NOTDISCARDED	no buffer has been discarded	0204
SLE_S_EOD	end of data reached	0205
SLE_S_NULL	the value of an object is NULL	0206
SLE_S_LOCKED	the object is currently locked and not accessible	0207
SLE_S_DEGRADED	the operation is performed albeit in degraded mode	0208
SLE_S_IGNORED	the request has been ignored as it is was not necessary	0209

B5 ERROR CODES DEFINED BY THIS RECOMMENDED PRACTICE (FACILITY ITF)

Mnemonic	Description	Code
SLE_E_STATE	a request is not valid in the current state of the server object	0200
SLE_E_PROTOCOL	a request cannot be accepted because it would cause an SLE protocol violation	0201
SLE_E_UNBINDING	an UNBIND operation has been invoked which the requester might not yet have noticed	0202
SLE_E_STOPPING	a STOP operation has been invoked, which the requester has not yet noticed	0203
SLE_E_ABORTED	the association has been aborted	0204
SLE_E_UNKNOWN	the object passed is not known	0205
SLE_E_INVALIDPDU	the PDU is not valid for the service type	0206
SLE_E_INVALIDID	an identifier is not valid	0207
SLE_E_BADVALUE	the value of an argument or operation parameter is not correctly formatted	0208
SLE_E_MISSINGARG	at least one argument or operation parameter is missing	0209
SLE_E_INCONSISTENT	the arguments or operation parameters are inconsistent	020A
SLE_E_RANGE	a value is out of the supported or specified range	020B
SLE_E_CONFIG	configuration data are incorrect, inconsistent, or cannot be supported	020C
SLE_E_OVERFLOW	general overflow condition	020D
SLE_E_SUSPENDED	the PDU cannot be accepted because data transfer is currently suspended	020E
SLE_E_DUPLICATE	invalid duplication of identifiers or objects	020F
SLE_E_NOFILE	the configuration file cannot be found or opened	0210
SLE_E_COMMS	a communications system failure occurred while processing a request	0211
SLE_E_TYPE	the type specification is not correct or arguments passed are not valid for the type	0212
SLE_E_PORT	a problem with the port identifiers exists	0213
SLE_E_TIME	the time specification is not correct or a request is not valid at this time	0214
SLE_E_SEQUENCE	general sequencing error	0215
SLE_E_UNSOLICITED	a response was received for which no request was issued	0216
SLE_E_ROLE	the requested role of the service instance is not supported or the request is not valid for the role of the service instance	0217
SLE_E_TIMERANGE	inconsistent specification of start and stop times of a period	0218
SLE_E_DIAGNOSTIC	unknown or inconsistent diagnostic codes found in a PDU	0219

ANNEX C

STRUCTURE OF THE SERVICE INSTANCE IDENTIFIER FOR VERSION 1 OF THE SLE SERVICES RAF, RCF, AND CLTU

(Normative)

C1 INTRODUCTION

C1.1 PURPOSE AND SCOPE

This annex provides a definition of the service instance identifier to be used by the SLE API components for version 1 of the SLE services RAF, RCF, and CLTU. This specification was used by implementations preceding this Recommended Practice, because the required definitions were not provided by version 1 of the CCSDS Recommended Standards for the SLE services RAF, RCF and CLTU (references [C1], [C2] and [C3]). For later versions of the SLE services RAF, RCF, and CLTU and for all other SLE service types, the specification in the CCSDS Recommended Standards for SLE transfer services shall be used.

The service instance identifier is a distinguished name as defined by reference [17], which is constructed according to the containment relationships of the managed objects in the CCSDS Recommended Standard for SLE service management. The attributes used in the service instance identifier, are the naming attributes of the managed objects. The values of the naming attributes are printable strings.

The SLE API only checks the validity of the attribute identifiers and does not check conformance to the containment relationships. These must be observed when the service instance identifier is defined. Therefore, this annex only provides a list of defined attributes and does not cover the containment relationships. The names of the managed objects and the names of their naming attributes are provided for information only. They are not processed by the API.

The API supports two formats for the service instance identifier:

a) the standard format as defined by reference [17], with the constraint that all attributes are character strings; and

b) a standard ASCII representation defined in this annex.

Conversion between the two formats is supported.

C1.2 REFERENCES

[C1] Space Link Extension—Return All Frames Service Specification, Draft Recommendation for Space Data System Standards, CCSDS 911.1-R-1.7, Red Book, Issue 1.7, September 1999.

[C2] *Space Link Extension—Return Channel Frames Service Specification,* Draft Recommendation for Space Data System Standards, CCSDS 911.2-R-1.7, Red Book, Issue 1.7, September 1999.

[C3] *Space Link Extension—Forward CLTU Service Specification,* Draft Recommendation for Space Data System Standards, CCSDS 912.1-R-1.99h, Red Book, Issue 1.99h, February 2000.

C2 STRUCTURAL ELEMENTS

C2.1 STANDARD FORMAT

The standard format consists of a sequence of 'attribute value assertions', i.e., pairs of an attribute identifier and an attribute value. The attribute identifier is an object identifier as defined by ASN.1 (reference [15]).

The detailed binary format is not defined by this Recommended Practice. It is implemented by the component SLE Utilities, which provides access to the format as defined by the interface `ISLE_SII` in annex A of this Recommended Practice.

C2.2 STANDARD ASCII REPRESENTATION

The standard ASCII representation makes use of human readable abbreviations for the attribute identifiers as defined in C3. The syntax of this representation is defined in C4.

C3 IDENTIFIERS AND ABBREVIATIONS FOR ATTRIBUTES

The Object IDentifiers (OID) and human readable abbreviations for attributes used in the service instance identifier are specified by the following table. References to the managed objects and the type names for attributes are provided for information only.

The object identifiers of the attributes differ only in the trailing component, which is identified in the table. The leading part of the object identifier is defined as:

```
{iso (1) identified-organization (2) ccsds (0) sle-services
        (9) service-management (5) attributes (2)}
```

This definition assumes that CCSDS will be registered by ISO directly below the node 'identified organization'. The number 0 is an arbitrary placeholder for the value that will be assigned to CCSDS.

As an example, the object identifier for the naming attribute of the managed object 'Forward CLTU Transfer Service Provided' (f-cltu-ts-p-mo-id) contains the following sequence of numbers:

1 2 0 9 5 2 7

Table C-1: Identifiers and Abbreviations for Attributes

Managed Object Class	Naming Attribute	OID	Abbreviation
DataStore	data-store-mo-id	1	ds
EventHandler	event-handler-mo-id	2	evh
F-AOS-SpaceLinkProcessing-FG	f-aos-space-link-processing-fg-mo-id	3	aos-fsl-fg
F-AOS-VC-DataInsertion-FG	f-aos-vc-data-insertion-fg-mo-id	4	aos-vcdi-fg
F-CLTU-Generation-FG	f-cltu-generation-fg-mo-id	5	cltugen-fg
F-CLTU-ST	f-cltu-st-mo-id	6	cltu-st
F-CLTU-TS-P	f-cltu-ts-p-mo-id	7	cltu
F-CLTU-TS-U	f-cltu-ts-u-mo-id	8	cltu-u
F-SpaceLinkAccessService	f-space-link-access-service-mo-id	9	fsl
F-SP-TS-P	f-sp-ts-p-mo-id	10	fsp
F-TC-F-ST	f-tc-f-st-mo-id	11	tcf-st
F-TC-F-TS-P	f-tc-f-ts-mo-id	12	tcf
F-TC-F-TS-U	f-tc-f-ts-u-mo-id	13	tcf-u
F-TC-SpaceLinkProcessing-FG	f-tc-space-link-processing-fg-mo-id	14	fsl-fg
F-TC-VCA-ST	f-tc-vca-st-mo-id	15	tcvca-st
F-TC-VCA-TS-P	f-tc-vca-ts-p-mo-id	16	tcvca
F-TC-VC-Channel-Prod	f-tc-vc-channel-prod-mo-id	17	ftcvc-prod
F-TC-VC-DataInsertion-FG	f-tc-vc-data-insertion-fg-mo-id	18	tc-vcdi-fg
F-VC-Seg-Prod	f-vc-seg-prod-mo-id	19	vcseg-prod
NotificationLog	notification-log-mo-id	20	nl
R-AF-ST	r-af-st-mo-id	21	raf-st
R-AF-TS-P	r-af-ts-p-mo-id	22	raf
R-AF-TS-U	r-af-ts-u-mo-id	23	raf-u
R-FrameDataExtraction-FG	r-frame-data-extraction-fg-mo-id	24	fde-fg

Managed Object Class	Naming Attribute	OID	Abbreviation
R-FrameProcessing-FG	r-frame-processing-fg-mo-id	25	fp-fg
R-MC-F-Prod	r-mc-f-prod-mo-id	26	mcf-prod
R-MC-FSH-ST	r-mc-fsh-st-mo-id	27	mcfsh-st
R-MC-FSH-TS-P	r-mc-fsh-ts-p-mo-id	28	mcfsh
R-MC-F-ST	r-mc-f-st-mo-id	29	mcf-st
R-MC-F-TS-P	r-mc-f-ts-p-mo-id	30	mcf
R-MC-OCF-Prod	r-mc-ocf-prod-mo-id	31	mcocf-prod
R-MC-OCF-ST	r-mc-ocf-st-mo-id	32	mcocf-st
R-MC-OCF-TS-P	r-mc-ocf-ts-p-mo-id	33	mcocf
R-MC-OCF-TS-U	r-mc-ocf-ts-u-mo-id	34	mcocf-u
R-MC-Prod	r-mc-prod-mo-id	35	mc-prod
R-MC-SHF-Prod	r-mc-shf-prod-mo-id	36	mcshf-prod
R-SpaceLinkAccessService	r-space-link-access-service-mo-id	37	rsl
R-SpaceLinkProcessing-FG	r-space-link-processing-fg-mo-id	38	rsl-fg
R-SP-ST	r-sp-st-mo-id	39	rsp-st
R-SP-TS-P	r-sp-ts-p-mo-id	40	rsp
R-VC-F-Prod	r-vc-f-prod-mo-id	41	vcf-prod
R-VC-FSH-Prod	r-vc-fsh-prod-mo-id	42	vcfsh-prod
R-VC-FSH-ST	r-vc-fsh-st-mo-id	43	vcfsh-st
R-VC-FSH-TS-P	r-vc-fsh-ts-p-mo-id	44	vcfsh
R-VC-F-ST	r-vc-f-st-mo-id	45	vcf-st
R-VC-F-TS-P	r-vc-f-ts-p-mo-id	46	vcf
R-VC-OCF-Prod	r-vc-ocf-prod-mo-id	47	vcocf-prod
R-VC-OCF-ST	r-vc-ocf-st-mo-id	48	vcocf-st
R-VC-OCF-TS-P	r-vc-ocf-ts-p-mo-id	49	vcocf
R-VC-OCF-TS-U	r-vc-ocf-ts-u-mo-id	50	vcocf-u
R-VC-Prod	r-vc-prod-mo-id	51	vc-prod
ServiceAgreement	service-agreement-mo-id	52	sagr
ServicePackage	service-package-mo-id	53	spack
SpacecraftChannelTree	spacecraft-channeltree-mo-id	54	sctree
SpacecraftCharacteristics	spacecraft-characteristics-mo-id	55	scchar
SpacecraftTracking	spacecraft-tracking-mo-id	56	sctrack

C4 SYNTAX OF THE STANDARD ASCII REPRESENTATION

The syntax of the standard ASCII representation is defined by the following grammar:

<service instance identifier> ::= <attribute-value pair>

['.' <attribute-value pair>]*

<attribute-value pair> ::= <attribute> '=' <value>

<attribute> is one of the abbreviations defined in table C-1

<value> is a string of printable characters

In addition, the following rules are applied:

a) All elements of the service instance identifier are not case sensitive with respect to comparisons.

b) The value of an attribute must not contain white space and must not contain the characters '.' and '='.

c) White space may be used between elements of the identifier, e.g., before and after '.' or '='.

Example:

The following example presents the identifier of a RAF service instance. The attribute values are arbitrary.

```
sagr=ESA-JPL-INTEGRAL.spack=routine-123.rsl=DL.rsl-fg=DL.raf=pass-21
```

ANNEX D

SIMPLE COMPONENT MODEL

(Normative)

D1 INTRODUCTION

The SLE API is based on the concept of integrating independently developed components with the SLE application. This concept has important advantages. For instance, it allows an organization offering SLE services to provide an API Proxy component to the users for integration into SLE user applications.

In order to simplify integration of independently developed components by a third party, dependencies between the components must be minimized. In addition, the need for delivery of source code and of the required building procedures should be avoided as far as possible. Finally, there must be some means of customizing a component for the specific environment in which it shall be deployed.

These objectives are supported by component models. However, at the time this Recommended Practice was developed, component systems were just emerging and a commonly accepted, platform-independent scheme was not available. Of the existing models, the Component Object Model (COM) developed by Microsoft (see reference [J5]) was the only one that could be directly used with the C++ language. However, COM requires a special run-time library and the presence of the COM Registry. Dependency on a special run-time environment that might not be readily available on all platforms was not considered acceptable for the SLE API.

Therefore, this Recommended Practice defines a very basic component model specifically for the SLE API. For this model it adopts a limited set of design patterns and conventions from COM. The conventions adopted are restricted to object interactions within the same address space and exclude detection and dynamic loading of components at runtime. The latter restriction implies that the COM library and the COM Registry are not needed.

The conventions and design patterns adopted from COM are:

a) Objects interact only via interfaces.

b) An interface is a collection of semantically related functions providing access to the services of an object. In C++, interfaces are specified by classes containing only public, pure virtual member functions. Interfaces can be derived from other interfaces.

c) Objects can implement more than one interface and support navigation between these interfaces. Interfaces are identified by a Globally Unique Identifier (GUID) assigned to every interface specification.

d) Interfaces are considered immutable once they have been published. If a modification must be applied, a new interface with a new identifier is created.

e) The lifetime of objects is controlled by reference counting. Methods to add a reference and to release a reference are provided by every interface.

f) To support navigation between interfaces and reference counting, every interface is derived from the interface `IUnknown`, specified by COM.

g) All non-object data structures passed across component boundaries must be allocated and de-allocated using a common memory manager that ensures consistency of dynamic memory management.

In addition to these conventions, this Recommended Practice also adopts the scheme for definition of result codes from COM.

It is stressed that components developed according to this Recommended Practice do not conform to COM. Important differences to COM are listed in subsection D6. However, it is possible to develop and use SLE API components in a COM environment. It is also possible to write very simple COM conforming wrappers for the components conforming to this Recommended Practice.

The Simple Component Model defined for the SLE API does not claim to cover all features that can be expected from a full scope component model. In particular it does not support:

a) detection and dynamic loading of components;

b) distribution of components to different processes and across a network;

c) event handling;

d) persistent storage;

e) inspection of components; and

f) customization of components.

For customization, this Recommended Practice uses the traditional concept of a configuration database, which is read by a component when a special method `Configure()` is called.

D2 COMPONENTS

Integration of independently developed components by a third party and substitutability is required only for the four API components identified in this Recommended Practice, namely:

a) the component API Proxy;

b) the component API Service Element;

c) the component SLE Operations; and

d) the component SLE Utilities.

Therefore, the term ‘component’ is only used for these modules. These components are of considerable size and complexity and support a rather large number of interfaces. It is assumed that each component makes use of several classes and creates a number of objects during operations. Some of these objects will implement functionality, which is not directly visible outside of the component, but others must be accessed by the application or by other components. In the following, objects that are visible outside of a component are referred to as ‘component objects’.

The design patterns and conventions for component objects ensure that the component implementing them can be easily integrated with other API components into an SLE Application program. However, there is no requirement that a component object itself be self-contained or be individually substitutable.

It must be stressed that the concept of a component object is constrained to the external view of a component. There is no requirement that a component object is actually implemented by a single C++ object. A single component object may be implemented by co-operation of several C++ objects and a single C++ object could provide the implementation of more than one component object. This Recommended Practice makes no assumptions on the internal design or implementation of a component.

All conventions defined in this annex only apply to component objects. Therefore, component objects are also referred to simply as ‘objects’ in this annex.

D3 DESIGN PATTERNS AND CONVENTIONS

D3.1 COMPONENT OBJECTS AND INTERFACES

A component object interacts with entities outside its component only via interfaces. An interface is a named set of semantically related functions providing access to the services of the component object. The client of an object only has access to the interface of a component object and never to the object itself.

In C++ an interface is defined by a class that only contains public, pure virtual function members and possibly constants. The interface must not contain any data members. Arguments of function members in an interface are restricted to:

a) basic C/C++ types;

b) C/C++ structures and arrays; and

c) references or pointers to other interfaces.

Functions in interfaces must not pass objects by value or reference of any class that is not an interface.

Interfaces may inherit from other interfaces. All interfaces are derived from the basic COM Interface `IUnknown`, specified in D5.2. Inheritance of interfaces is restricted to single inheritance.

NOTE – It is stressed that the restriction of inheritance to a single base class only applies to interfaces and not to implementation classes.

C++ classes implement interfaces by inheriting the interface and providing an implementation for each of the inherited function members.

D3.2 INTERFACE IDENTIFIERS AND INTERFACE VERSIONS

An interface is identified by an Interface ID (IID). The IID is a 'Globally Unique ID' (GUID), which is a 128 bit binary value generated from a 48-bit unique machine identifier and UTC time. Its structure is shown in D5.1, which also contains some hints on how it can be easily handled.

Following COM, an interface is defined to be immutable; i.e., once an interface is published it is never changed. If an interface must be modified, or the service provided by the interface changes, it is replaced by a new interface with a different IID. An object may or may not continue to support the old interface, but all new functionality is provided only via the new interface. If the old interface is no longer supported, clients requesting the interface with the old IID will receive an error and will thus be able to detect incompatibilities.

D3.3 MULTIPLE INTERFACES OF OBJECTS

An object may provide more than one interface. Objects providing multiple interfaces support navigation between interfaces via the method `QueryInterface()` defined in the interface `IUnknown`. A client holding a reference to one of the interfaces implemented by an object asks for a different interface presenting the IID of that interface. If the object also implements that interface, it returns a pointer to it. Otherwise it returns an error.

In C++, multiple interfaces per object can be implemented using multiple inheritance, or by referencing a special object for every interface. These implementation methods are described in 3.6 of the COM Specification. It refers to the latter option as 'Interface Containment'.

It is required that any query for the specific interface `IUnknown` always returns the same actual pointer value, no matter through which interface derived from `IUnknown` `QueryInterface()` is called. This requirement does not apply to other interfaces of the object. Therefore, the pointer to `IUnknown` serves as the only unique identifier of the object itself.

D3.4 OBJECT LIFETIME AND REFERENCE COUNTING

Because clients only receive references to an interface of an object and never to the object itself, they cannot create objects by standard C++ means. A client can receive a reference to an interface as the return value of a method called on another interface or as an output argument of a method call.

In contrast to COM, this Recommended Practice does not include any other method to obtain an interface, in particular it does not support class factories or the COM library function `CoCreateInstance()`. For each of the four API components, a bootstrap 'creator function' is defined providing a reference to a specific interface implemented by the component. Clients can obtain further interfaces using `QueryInterface()`, or can request creation of objects via special 'factory interfaces' defined in the API.

The lifetime of an object is determined by a reference count, which is controlled by the methods `AddRef()` and `Release()` defined in `IUnknown`. Whenever a client obtains a reference to an interface, it calls `AddRef()` on that interface, incrementing the reference-count. When a client no longer needs the interface it calls `Release()` on the interface, decrementing the reference-count. When all reference-counts for all interfaces of an object are zero, the object is automatically deleted. Clients must never invoke the delete operator on interfaces.

In addition, the following conventions apply:

a) `QueryInterface()` automatically calls `AddRef()` before returning the interface pointer, such that the caller should not call `AddRef()`. The client obtaining the interface pointer must call `Release()`, however.

b) Objects are usually created with a reference count of zero and the creating function (e.g., a method of a factory interface) calls `QueryInterface()` to set the reference count to one. While this is only one implementation option, all functions creating objects must ensure that the reference count is one after creation. For the client, the statements made for `QueryInterface()` apply.

Clients must not make any assumptions on how an object is implemented, and must strictly call `AddRef()` and `Release()` on every interface. Implementation of the reference count depends on the method used for implementation of interfaces. Objects may use a reference-count per interface or a single reference count per object. In a multi-threaded environment, the methods `AddRef()` and `Release()` must be implemented in a MT-safe manner.

Further applicable rules for reference counting are defined in the COM Specification (see 3.3.2). It is stressed that adherence to these rules must be carefully verified, because any failure to do so implies the danger of memory leaks or premature deletion of objects with unpredictable effects.

D3.5 MEMORY MANAGEMENT

Components are free to use any memory management scheme for their internal implementation. Depending on the platform, use of different, incompatible memory managers by different components is not uncommon. In order to ensure that data structures passed across component boundaries can be allocated by one component and safely de-allocated by another component, COM provides a specific memory manager that must be used for such data. This memory manager implements the interface `IMalloc`, which can be obtained from the COM runtime.

SCM adopts this approach in principle but assigns the implementation of the memory manager used for the SLE API to the component API Utilities. In order to allow use of the COM memory manager in a COM environment, the SLE API memory manager provides the interface `IMalloc` as well (see A4.3). However, the interface `IMalloc` includes methods that cannot be easily provided with the default memory management features available on other platforms, and implementation of the component API Utilities are not required to implement them. This applies to the methods `GetSize()`, `DidAlloc()`, and `HeapMinimize()`. Implementations may provide dummy implementations and clients must not rely on these methods.

It is pointed out that the API supplied memory manager need not be used for objects implementing interfaces as consistent memory management is ensured by reference counting in this case.

The pointer to `IMalloc` shall be obtained by calling the method `CreateMemoryManager()` of the Utility Factory.

D3.6 RESULT CODES

This Recommended Practice adopts the COM conventions for result codes returned by interface methods. The result codes defined for the SLE API are specified in annex B.

D4 CONVENTIONS FOR SLE APPLICATIONS

Applications programs using the SLE API must adhere to the conventions defined in this annex as clients of SLE API components. In particular, correct functioning of the API can only be ensured if applications handle reference counting correctly and use the memory manager supplied by the API for data structures passed to the API and received from the API.

For reasons of consistency, the interfaces that must be implemented by applications follow the same rules as interfaces provided by the API. This implies that the interface `IUnknown` must be fully supported. However, this Recommended Practice does not define any component objects with multiple interfaces that need to be implemented by an application.

Therefore, navigation needs only be supported between an interface provided by the application and `IUnknown`.

Applications must ensure that objects providing an interface for use by the SLE API are not deleted as long as any API component still holds a reference to the interface. This requirement can be met by implementing the reference counting scheme as described in D3.4. Applications are not required to delete an object when the reference count becomes zero if they use other means to handle object lifetimes.

D5 INTERFACE DEFINITION AND IDENTIFICATION

D5.1 INTERFACE IDENTIFIERS

The Globally Unique Identifier is specified by the following C structure (defined in `SLE_SCMtypes.h`):

```
typedef struct GUID
{
  unsigned long  Data1;
  unsigned short Data2;
  unsigned short Data3;
  unsigned char  Data4[ 8 ];
} GUID;
```

D5.2 THE INTERFACE IUNKNOWN

The GUID for the interface `IUnknown` is defined as

`{00000000-0000-0000-C000-000000000046}`

The formal specification of the interface is provided below. It can be found in the file `SLE_SCM.H`.

Synopsis

```
#include "SLE_RESULT.h"
#define IID_IUnknown_DEF { 0x00000000, 0x0000, 0x0000, \
          { 0xc0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46 } }

interface IUnknown
{
  virtual HRESULT
    QueryInterface( const GUID& iid, void** ppv ) = 0;
  virtual unsigned long
    AddRef() = 0;
  virtual unsigned long
    Release() = 0;
};
```

Methods

```
HRESULT QueryInterface( const GUID& iid, void** ppv );
```

Returns a pointer to the interface identified by the argument iid if that is supported by the object. If the object on which the function is called does not support the interface, returns the error code `E_NOINTERFACE`.

```
unsigned long AddRef();
```

Increments the reference-count for the interface on which it is called, and returns the value of the count.

```
unsigned long Release();
```

Decrements the reference-count for the interface and returns the new value. When the reference count for all interfaces of an object becomes zero, the object is deleted.

D6 DIFFERENCES TO COM

It is stressed once more that this Recommended Practice does not imply use of COM. It also does not claim to define a complete object model. It only adopts a very limited subset of the COM conventions. The following are major differences to COM:

a) Dynamic loading and linking of component servers at runtime is not supported. All components must have been linked with the program using them. Therefore, the SLE API does not require the COM library, or the COM Registry, or any other special runtime environment.

b) Because all components must be linked with the program using them it requires use of unique names at global scope within a component.

c) The Class Factory and the global function `GetClassObject()` are not supported.

d) All interactions between API components are local to one address space. Use of remote procedure calls between components is not foreseen. This Recommended Practice does not exclude that parts of a component actually reside in a different address space.

e) The interface `IMalloc` is provided by the component API Utilities and might only provide dummy implementations for the methods `GetSize()`, `DidAlloc()`, and `HeapMinimize()`.

f) The use of a COM-specific memory manager for allocation and release of memory objects passed over interfaces is required only in certain cases (see A2.6). Only a subset of the methods of the `IMalloc` memory manager interface are supported by this Recommended Practice.

g) This Recommended Practice does not foresee use of any interfaces specified by COM other than `IUnknown` and `IMalloc`.

D7 WORKING IN A COM ENVIRONMENT

The API defined in this Recommended Practice may be implemented in a COM based environment or integrated into an application running in a COM environment. In this case, the specifications adopted from COM in the header files `SLE_SCM.H`, `SLE_SCMTypes.h` and `SLE_RESULT.h` should be removed and replaced by an inclusion of the original COM files.

Obviously, standard COM behavior cannot be expected from a component developed according to this Recommended Practice. However, a COM conforming wrapper may be easily provided by implementing a Class Factory, which uses the specific bootstrap 'creator function' defined in this Recommended Practice.

ANNEX E

CONFORMANCE

(Normative)

E1 INTRODUCTION

The conformance requirements defined in E2 aim at two objectives, which are partially conflicting:

a) ensuring a maximum of compatibility between independently developed components to support substitutability; and

b) supporting a wide range of possible implementations, which are tailored to the actual needs of the systems in which the API is used.

In order to support substitutability, the range of options must be minimized. On the other hand, this Recommended Practice defines an API that supports all possible requirements an application may have, and the need for more restricted implementations is recognized. Examples of limited implementations include:

a) a 'lightweight' API for SLE service users, which only need to initiate BIND operations;

b) a specialized API for SLE service providers, which never act as service users.

Developers might also want to constrain the range of support provided for different configurations of processes, for use of in-process threads, or for diagnostics.

Therefore, this Recommended Practice defines a number of optional features for the components API Proxy (see E3.1) and API Service Element (see E4.2). It must be stressed, however, that every reduction in the scope of features provided by an implementation reduces the scope of environments in which a component can be used. Implications of not supporting an optional feature are described for each of the options.

Implementations may provide features that are not defined in this Recommended Practice and may define additional interfaces for access to these features. However, a conforming implementation must not require use of such additional features or interfaces nor depend on additional features or interfaces provided by other components.

This Recommended Practice does not define requirements with respect to the numbers of service instances, associations, etc., an implementation must support. Subsection E5 identifies parameters for which an implementation might impose limits. Such constraints should be clearly specified for a conforming implementation.

In a strict sense, conformance requirements only apply to API components, and not to the SLE Application. However, use of the API requires that the application provides the required interfaces and uses the API as specified. Related requirements can be found in E6.

E2 CONFORMANCE REQUIREMENTS

Software products claiming conformance to this Recommended Practice must provide an implementation of one or more of the components API Proxy, API Service Element, SLE Operations, and SLE Utilities. For components implemented by the product, the following conformance requirements apply:

C-1 The component API Proxy must conform to all specifications in 3.2, 3.7 and 4.4, except for the options identified in E3.1 and the backward compatibility features identified in E4. In addition, the component must adhere to the rules for use of interfaces supplied by the application as defined in 3.6.

C-2 For the component API Proxy, any constraints with respect to the parameters identified in E5.1 must be specified.

C-3 The component API Service Element must conform to all specifications in 3.3, 3.7 and 4.5, except for the options identified in E4.2 and the backward compatibility features identified in E4. In addition, the component must adhere to the rules for use of interfaces supplied by the application as defined in 3.6.

C-4 For the component API Service Element, any constraints with respect to the parameters identified in E5.2 must be specified.

C-5 The component SLE Operations must conform to all specifications in 3.4, except for the backward compatibility features identified in E4.

C-6 The component SLE Utilities must conform to all specifications in 3.5, except for the backward compatibility features identified in E4.

C-7 Components must provide all interfaces specified for the component in annex A, unless the interface explicitly refers to an option that is not provided by the component.

C-8 The components API Proxy, API Service Element, and SLE Operations must support at least one SLE transfer service type. For every supported service type the components must conform to the specifications in the relevant supplemental Recommended Practice for the service-specific API.

C-9 Components must be able to provide all features defined in this Recommended Practice using the services of the other components defined in this Recommended Practice via the interfaces defined in annex A, and must not require any additional services or interfaces.

C-10 All components must adhere to the conventions defined in annex D to this Recommended Practice.

C-11 It must be possible to use every component provided by a product individually in combination with other components provided by a different conforming product, with the restrictions identified in E3 for certain options.

C-12 If a component claims to implement an option defined in E3, the implementation must fully conform to this Recommended Practice.

C-13 If a component claims to support backward compatibility as specified in E4, the implementation must fully conform to this specification.

E3 OPTIONS

E3.1 API PROXY

E3.1.1 Overview

The optional features defined for the component API Proxy are summarized in table E-1.

Table E-1: Optional Features for the API Proxy

ID	Option	Remarks / Reference
1	Initiator role for associations	At least one of the two options must be supported (see E3.1.2)
	Responder role for associations	
2	Support for ‘sequential flows of control’ and support for the interface `ISLE_Sequential`	At least one of the two options must be supported (see E3.1.3)
	Support for ‘concurrent flows of control’ and support for the interface `ISLE_Concurrent`	
3	Operation mode for a gateway	see E3.1.4
4	Routing of BIND invocations to different processes	see E3.1.5
5	Diagnostic traces	see E3.1.6

E3.1.2 Roles in the BIND Operation

E3.1.2.1 Options

An implementation of the component API Proxy shall support one of the following options:

PXO-1a Associations in the initiator role as specified in 3.2.4.2.1.

PXO-1b Associations in the responder role as specified in 3.2.4.2.2.

PXO-1c Associations in the initiator role and associations in the responder role.

E3.1.2.2 Implications

A proxy that does not support associations in the initiator role cannot be used by an application that needs to initiate BIND operations.

A proxy that does not support associations in the responder role cannot be used by an application that needs to respond to BIND invocations.

E3.1.3 Handling of Multiple Flows of Control

E3.1.3.1 Options

An implementation of the component API Proxy shall support one of the following options:

PXO-2a Support for ‘sequential flows of control’ specified in 3.2.10.2 and 3.7.2 as well as the associated control interface `ISLE_Sequential`, defined in A6.1.1.

PXO-2b Support for ‘concurrent flows of control’ specified in 3.2.10.2 and 3.7.3 as well as the associated control interface `ISLE_Concurrent`, defined in A6.1.6.

PXO-2c Both the ‘sequential’ and the ‘concurrent’ options and the associated control interfaces.

If a proxy supports option PXO-2c, the implementation shall define the means by which the interface behavior is selected.

E3.1.3.2 Implications

A proxy supporting only ‘sequential flows of control’ cannot be used by clients that do not support this behavior for the interface to the proxy. The proxy cannot be used by clients that do not provide the interfaces `ISLE_EventMonitor` and `ISLE_TimerHandler`.

A proxy supporting only the interface for ‘concurrent flows of control’ cannot be used in a single-threaded environment or by clients that do not support that behavior.

E3.1.4 Operation Mode for a Gateway

E3.1.4.1 Options

The following feature is optional for a proxy implementation:

PXO-3 The ‘pass-through mode of operation’ specified in 3.2.7.

If a proxy supports option PXO-3, the implementation shall define the means by which the pass-through mode is enabled.

E3.1.4.2 Implications

A proxy not supporting this option cannot be used within a gateway.

E3.1.5 Routing of BIND Invocations to Processes

E3.1.5.1 Options

A conforming proxy implementation must support a configuration in which the hosting process handles all service instances for which BIND invocations are received on one or more ports. The following configuration is an optional feature:

PXO-4 Support for a configuration in which service instances using one or more ports are distributed to existing processes in a manner defined by the application.

E3.1.5.2 Implications

A proxy not supporting this option cannot be used in a system that requires the associated support.

E3.1.6 Diagnostic Traces

E3.1.6.1 Options

The following feature is optional:

PXO-5 Support for diagnostic traces as defined in 3.2.9 and 3.6.3.

A proxy not supporting this option must respond with `E_NOINTERFACE` when the interface `ISLE_TraceControl` is requested via a call to `QueryInterface()` on the proxy or on an association object.

E3.1.6.2 Implications

Traces are not available if a proxy does not support this option. Tracing by a proxy is constrained because tracing of individual associations is not possible. Simultaneous tracing of all associations can still be controlled directly via the interface exported by the API Proxy component itself.

E4 BACKWARD COMPATIBILITY

E4.1 GENERAL

Features specified in this specification only to support version 1 of the SLE transfer services RAF, RCF, and CLTU are not considered mandatory. However, API components that do not implement these features must nevertheless provide exactly the interfaces specified in annex A of this specification. The same rule applies to the supplemental Recommended Practice documents for the service-specific APIs for the services RAF, RCF, and CLTU.

E4.2 API SERVICE ELEMENT

E4.2.1 Overview

The optional features defined for the component API Service Element are summarized in table E-2.

Table E-2: Optional Features for the API Service Element

ID	Option	Remarks / Reference
1	User role for service instances	At least one of the two options must be supported (see E4.2.2)
	Provider role for service instances	
2	Provider-initiated binding	see 1.2.2 item b)
3	Support for ‘sequential flows of control’ and support for the interface `ISLE_Sequential`	At least one of the two options must be supported (see E4.2.3)
	Support for ‘concurrent flows of control’ and support for the interface `ISLE_Concurrent`	
4	Diagnostic traces	see E4.2.4

E4.2.2 SLE Service User and Provider Roles

E4.2.2.1 Options

An implementation of the component API Service Element shall support one of the following options:

SEO-1a Service instances for use by an SLE service user application (‘user role’).

SEO-1b Service instances for use by an SLE service provider application (‘provider role’).

SEO-1c Service instances for use by an SLE service user application and service instances for use by an SLE service provider application.

E4.2.2.2 Implications

A service element that does not support service instances in the user role cannot be used by an application that must act as an SLE service user.

A service element that does not support service instances in the provider role cannot be used by an application that must act as an SLE service provider.

E4.2.3 Handling of Multiple Flows of Control

E4.2.3.1 Options

An implementation of the component API Service Element shall support one of the following options:

SEO-2a Support for ‘sequential flows of control’ specified in 3.3.8.2 and 3.7.2 as well as the associated control interface `ISLE_Sequential`, defined in A6.1.1.

SEO-2b Support for ‘concurrent flows of control’ specified in 3.3.8.2 and 3.7.3 as well as the associated control interface `ISLE_Concurrent`, defined in A6.1.6.

SEO-2c Both the ‘sequential’ and the ‘concurrent’ options and the associated control interfaces.

If an implementation supports option SEO-2c the implementation shall define the means by which the interface behavior is selected by the application.

A service element may provide different options for the interface to the application and for the interface to the proxy.

E4.2.3.2 Implications

A service element supporting only ‘sequential flows of control’ on the application interface cannot be used by applications that do not support this behavior. The service element cannot be used by applications that do not support the interfaces `ISLE_EventMonitor` and `ISLE_TimerHandler`.

A service element supporting only ‘sequential flows of control’ on the proxy interface, cannot use a proxy that does not support this behavior.

A service element supporting only ‘concurrent flows of control’ on the application interface cannot be used in a single-threaded environment or by applications, that do not support that behavior.

A service element supporting only ‘concurrent flows of control’ on the proxy interface cannot use a proxy not supporting that behavior.

E4.2.4 Diagnostic Traces

E4.2.4.1 Options

The following feature is optional:

SEO-3 Support for diagnostic traces as defined in 3.3.7 and 3.6.3.

A service element not supporting this option must respond with `E_NOINTERFACE` when the interface `ISLE_TraceControl` is requested via a call to `QueryInterface()` on the service element or on a service instance object.

E4.2.4.2 Implications

Traces are not available if a service element does not support this option. Tracing by a proxy is constrained because tracing of individual associations is not possible. Simultaneous tracing of all associations can still be controlled directly via the interface exported by the API Proxy component itself.

E5 LIMITS

E5.1 API PROXY

An implementation of the component API Proxy may constrain the values of the parameters identified in table E-3, as long as the minimum value indicated in the table is supported. The minimum value is required to support any useful operation; it is expected that implementations will generally support higher numbers.

Table E-3: Parameters That May Be Constrained by a Proxy

Parameter		Minimum
1	Maximum number of concurrent bound associations per process	1
2	Maximum number of concurrent bound associations in total on a system	1
3	Maximum number of ports used for outgoing BIND invocations per process.	1
4	Maximum number of ports on which an incoming BIND invocation can be received per process	1
5	Maximum number of ports on which an incoming BIND invocation can be received in total on a system	1
6	Maximum number of incoming PDUs that can be queued (per association / per process)	1/1
7	Maximum number of incoming TRANSFER-DATA invocations and TRANSFER-BUFFER invocations that can be queued (per association / per process)	1/1
8	Maximum number of outgoing PDUs that can be queued	1
9	Maximum number of pending remote returns per association	1
10	Maximum size of a PDU	100 KB

E5.2 API SERVICE ELEMENT

An implementation of the component API Service Element component may constrain the values of the parameters identified in table E-4, as long as the minimum value indicated in the table is supported. The minimum value is required to support any useful operation; it is expected that implementations will generally support higher numbers.

Table E-4: Parameters That May Be Constrained by a Service Element

Parameter		Minimum
1	Maximum number of service instances that can exist concurrently	1
2	Maximum number of concurrently bound service instances	1
3	Maximum number of proxies that can be supported concurrently	1
4	Maximum number of pending remote returns per service instance	1
5	Maximum number of pending local returns per service instance	1

E6 SLE APPLICATIONS

E6.1 GENERAL

SLE Applications are not part of the SLE API, and therefore not subject to conformance requirements. However, use of the API requires the application to conform to the specifications in 3.6 and 3.7 and to provide the interfaces defined in A9, with exception of the options defined in E6.2 and E6.3.

E6.2 HANDLING OF MULTIPLE FLOWS OF CONTROL

An application shall support the interface behavior ‘sequential’ or ‘concurrent’, or both. It must be able to handle the control interface associated with the supported behavior.

E6.3 DIAGNOSTIC TRACES

An application is not required to provide the interface `ISLE_Trace`. If that interface is not provided, that application must not invoke any of the methods defined by the interface `ISLE_TraceControl` (see A6.2).

ANNEX F

INTERACTION OF COMPONENTS

(Informative)

F1 INTRODUCTION

This annex displays a set of diagrams providing an overview on how the components specified by the model are configured and how they interact. It does not contain any new specifications.

Overview diagrams in subsection F2 show configuration of components and use of interfaces within the complete API. Subsection F3 provides sequence diagrams and collaboration diagrams for selected scenarios, including:

a) initialization, start-up, and shutdown of the API;

b) use of operation objects;

c) execution of the BIND operation on the SLE user side (user-initiated binding);

d) execution of the UNBIND operation on the SLE user side (user-initiated binding);

e) execution of the BIND operation on the SLE provider side (user-initiated binding);

f) execution of the UNBIND operation on the SLE provider side (user-initiated binding).

The purpose of these scenarios is to explain the general concepts, not to specify details concerning the objects involved and the actual interfaces and methods called. Therefore, the diagrams display ‘conceptual objects’ and ‘conceptual messages’ and do not reference the objects of the model directly. It should, however, be straightforward to map these scenarios to the interfaces and methods specified. The interfaces used are referenced in the text where that has been felt to be necessary.

F2 CONFIGURATION OF COMPONENTS AND INTERFACES

Figure F-1 shows how the components API Proxy and API Service Element are configured and how they are used by the application. The relationships shown in this diagram are actually a ‘shortcut’ of the actual relationships. Cross-component references are always references to interfaces not to ‘component classes’.

This specification assumes that a single instance of the component class API Service Element is used by an application. The service element is linked with one or more instances of the component class API Proxy. Individual instances used by the service element are identified by the attribute ‘Protocol ID’, which identifies the technology and mapping used by the

proxy. The service element routes outgoing BIND invocations to the proxy supporting the required protocol ID. It derives the protocol ID from a table in its configuration database that maps port identifiers to protocol IDs.

The service element manages instances of the component class API Service Instance. These instances are created and deleted by the application. The maximum number of these instances is not constrained by this specification, but may be constrained by an implementation. During its complete lifetime, an object of the class API Service Instance is associated with exactly one object of the class SLE Application Instance. In periods, in which data are exchanged between an SLE user and an SLE provider, an object of the class API Service Instance uses exactly one object of the class Association. During its lifetime, an object of the class SLE Application Instance may be using several different instances of component class Association.

An overview of the most important interfaces exported by components and used by other components is shown in figure F-3.

Figure F-2 provides further details on the configuration of interfaces used for service provisioning. All these interfaces provide methods to pass operation objects for invocations (`InitiateOpInvoke()` and `InformOpInvoke()`) and returns (`InitiateOpReturn()` and `InformOpReturn()`). 'Initiate' interfaces are used for invocations and returns issued locally. The complementary 'Inform' interfaces are used to pass invocations and returns received from the peer system. It should be noted, however, that the diagram does not show all details; individual interfaces provide special methods needed only for the specific interface.

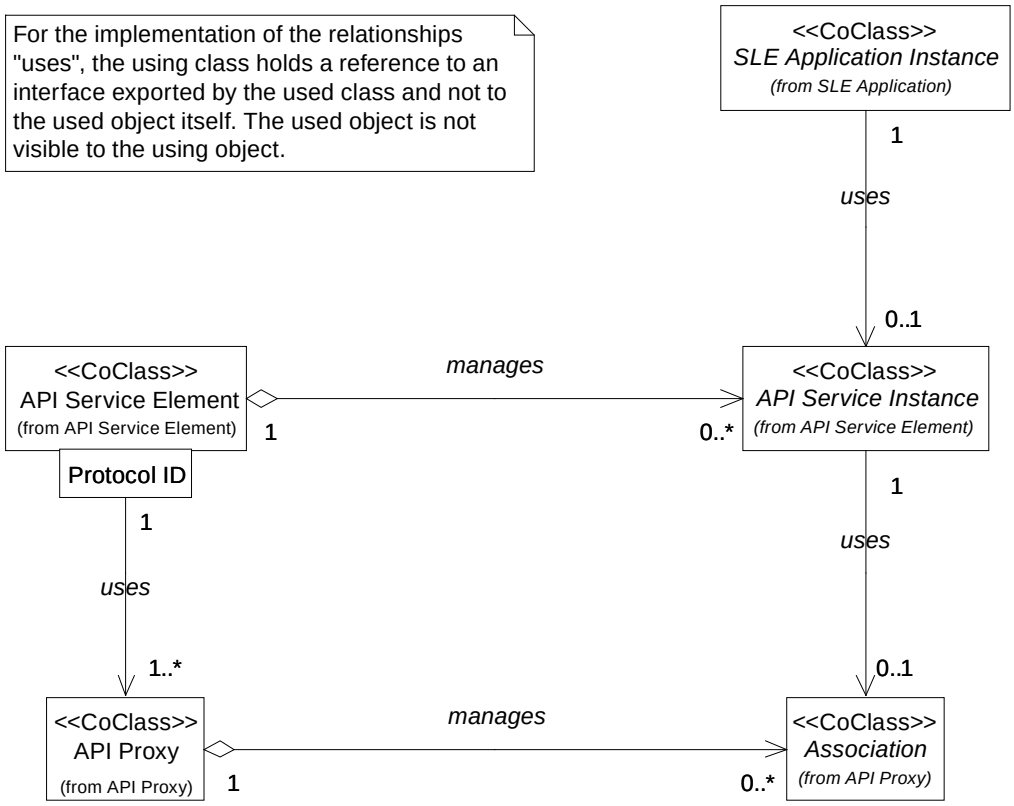

Figure F-1: Configuration of Components

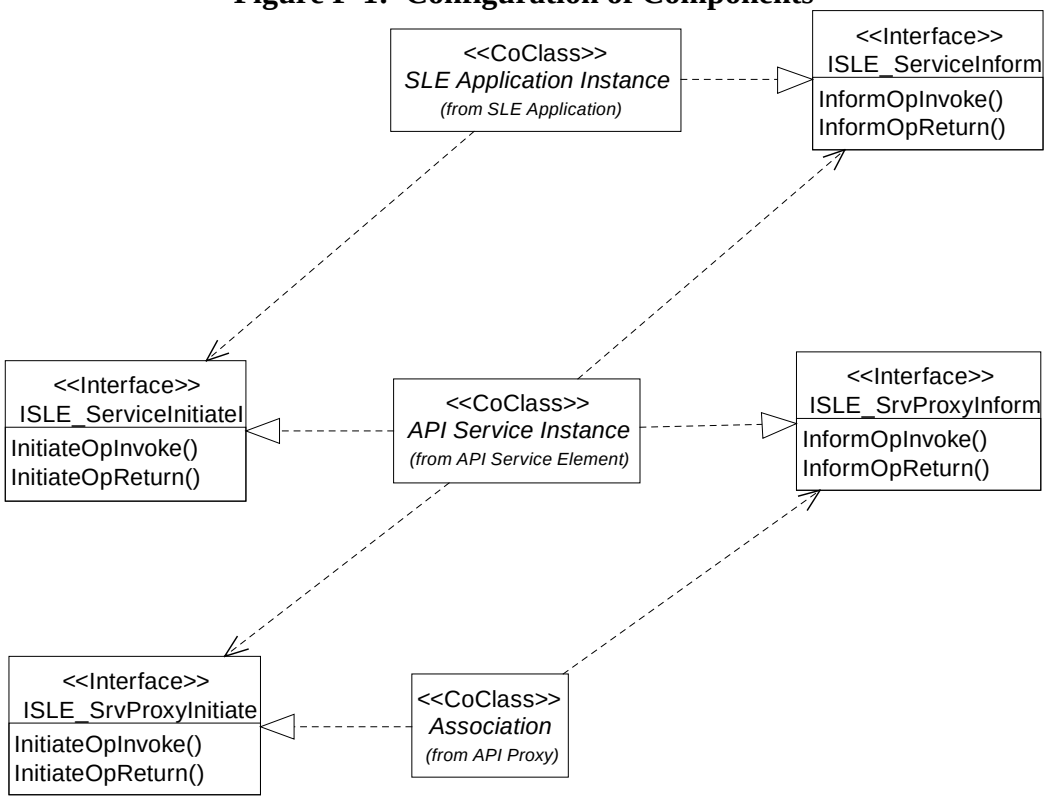

Figure F-2: Configuration of Interfaces for Service Provisioning

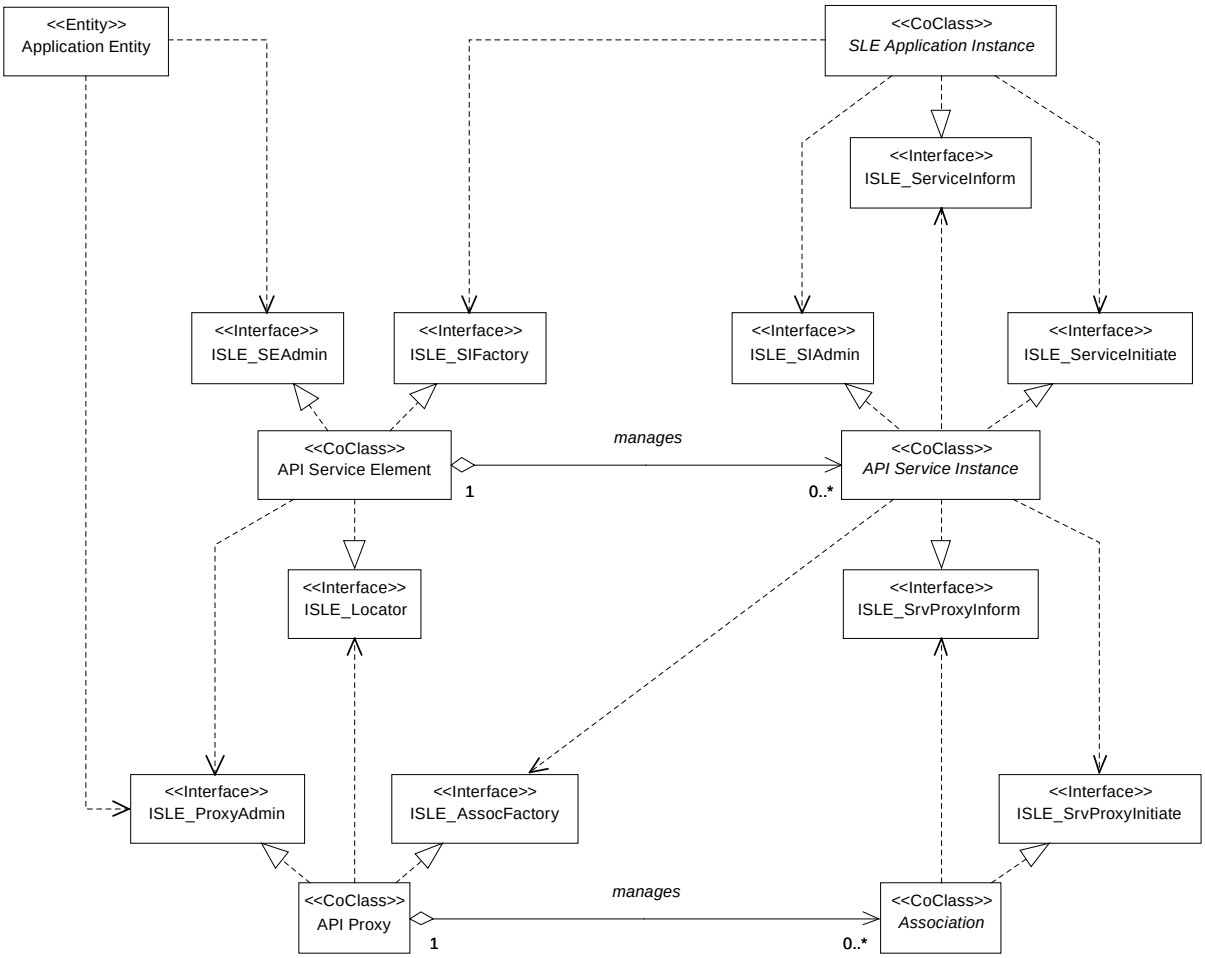

Figure F-3: Interaction of API Components

F3 SCENARIOS

F3.1 INITIALIZATION AND SHUTDOWN OF THE API

F3.1.1 General

Figure F-4 shows the sequence of actions that are required for start-up and shutdown of the API.

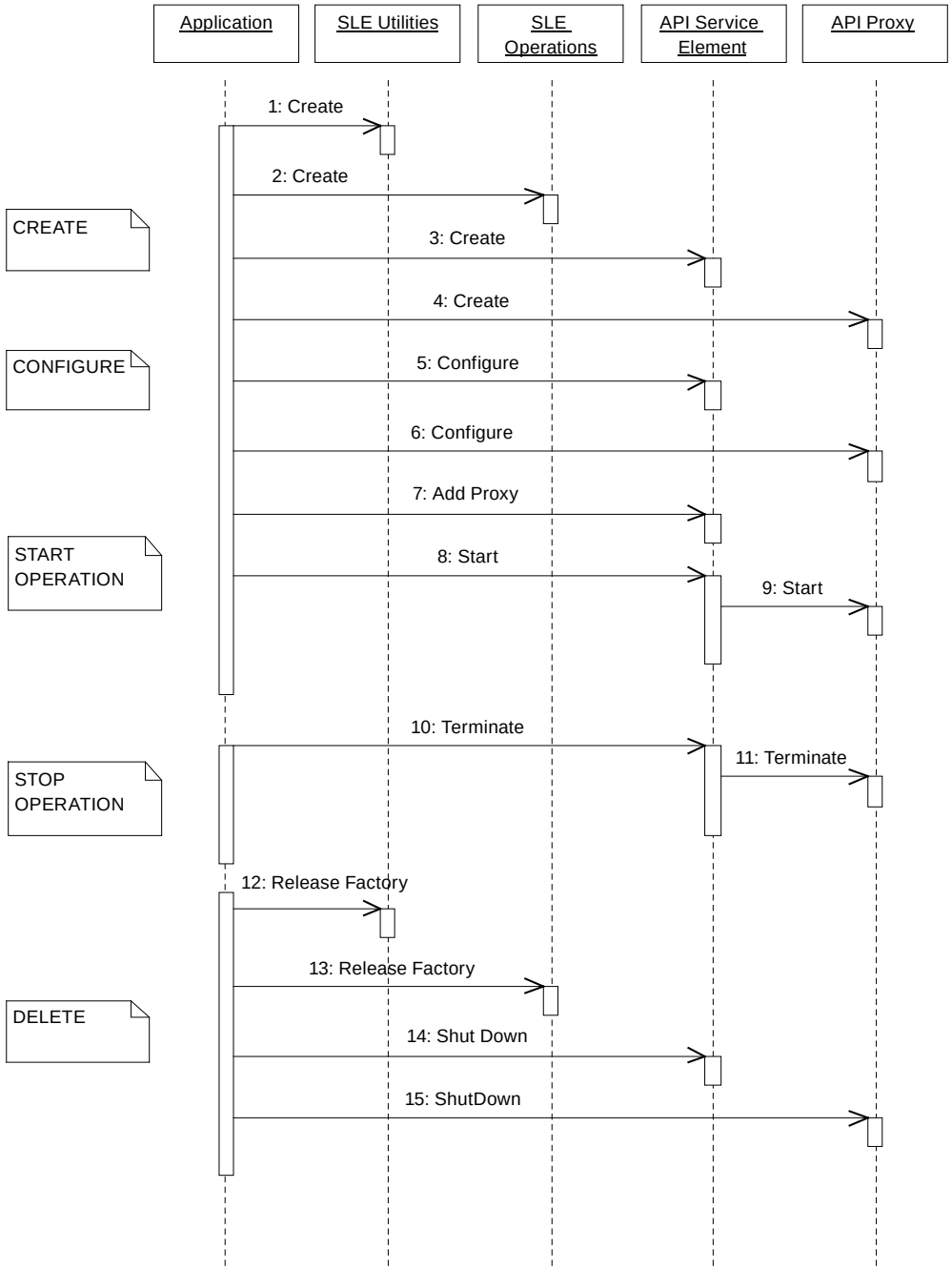

Figure F-4: Initialization and Shutdown

Start-up of the API is performed in three stages: creation; configuration; start of operation. Orderly shutdown of the API is performed in two steps: stop of operation; deletion.

F3.1.2 Creating the API

The application must create instances of every API component by a call to the creator function of the component (steps 1-2-3-4). The sequence of these steps is generally not important, with the exception that creation of the component SLE Operations requires a reference to the SLE Utility Factory as input argument. If more than one proxy is used, each of these proxies must be created.

F3.1.3 Configuring the API

The application must configure all instances of the components API Service Element (step 5) and API Proxy (step 6), passing the name of the configuration file for the component and references to interfaces of the other components. Configuration is completed by linking the API Service Element with all instances of the component API Proxy (step 7).

F3.1.4 Starting the API

After successful configuration, the application must start operation of the API by a call to the start method of the API Service Element (step 8). The API Service Element then starts all proxies with which it has been linked (step 9). Depending on the configuration, the API will now start listening for incoming connect requests and will accept creating and binding of service instances.

F3.1.5 Stopping the API

To stop operation of the API, the application calls the terminate method of the API Service Element (step 10), which in turn calls the terminate method of all proxies which it has started (step 11). The terminate-method aborts all current activities. After completion of the method, the API is in the state it was after configuration.

F3.1.6 Deleting the API

To release all resources held by the API and to delete the API components, the application must release all references it may still hold on API interfaces. In particular it must release the operation object factory interface and the utility factory interface (steps 12-13). The application then calls the method `ShutDown()` on the service element and all proxy instances (steps 14-15). As part of the method `ShutDown()` the service element and the proxy release all references to interfaces of other components they hold, delete all internal objects, and release all resources.

F3.2 USE OF OPERATION OBJECTS

F3.2.1 General

Use of operation objects within the API is shown by the collaboration diagram in figure F-5 and the sequence diagram in figure F-6. These two diagrams are identical with respect to the contents; they only differ in the way the scenario is presented.

The scenario presents processing of a confirmed operation by the API. The operation is requested by the INVOKER on the left-hand side of the diagrams and performed by the PERFORMER on the right-hand side. Objects on the invoker side are marked by '(I)' and objects on the performer side by '(P)'.

To invoke an operation, the application creates an operation object of the required type using the interface `ISLE_SIOpFactory` provided by the service instance (step 1). It could also use the operation object factory, but the service instance provides the additional service to initialize the parameters according to its own configuration.

The application then sets the invocation parameters using the interface provided by operation object (step 2) and passes a reference to the operation object to the service instance via the interface `ISLE_ServiceInitiate` (step 3).

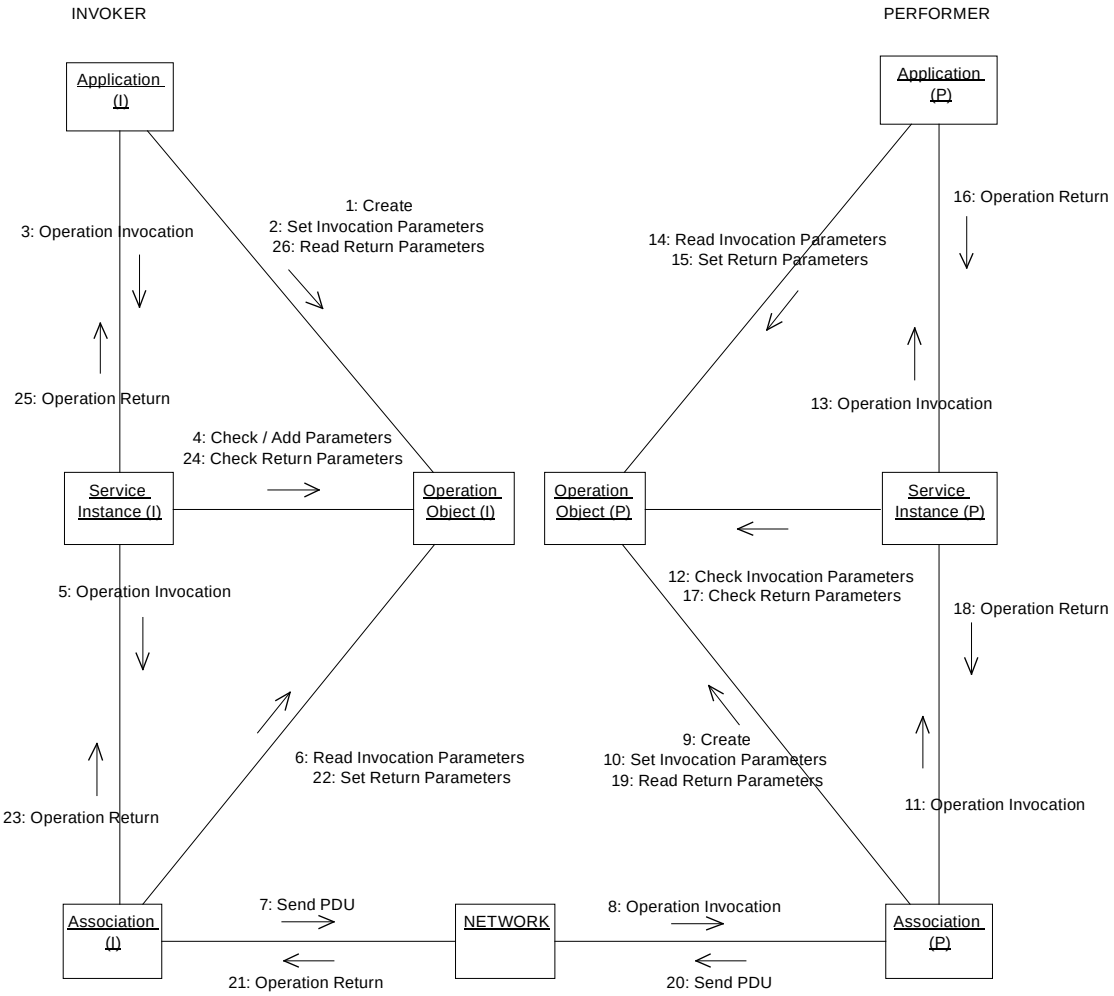

Figure F-5: Collaboration Diagram for Use of Operation Objects

The service instance verifies the validity of the invocation in its current state, checks the parameters, adds the parameters it handles, places the reference on a list of pending remote returns, and passes the reference to the association via the interface `ISLE_SrvProxyInitiate` (steps 4-5). It also starts a return timer and associates that with the operation object.

The association on the invoker side reads the invocation parameters from the operation object, encodes the data, constructs the PDU, and transmits it to the peer proxy on the performer side (steps 6-7). The association memorizes the operation object on a list of pending returns.

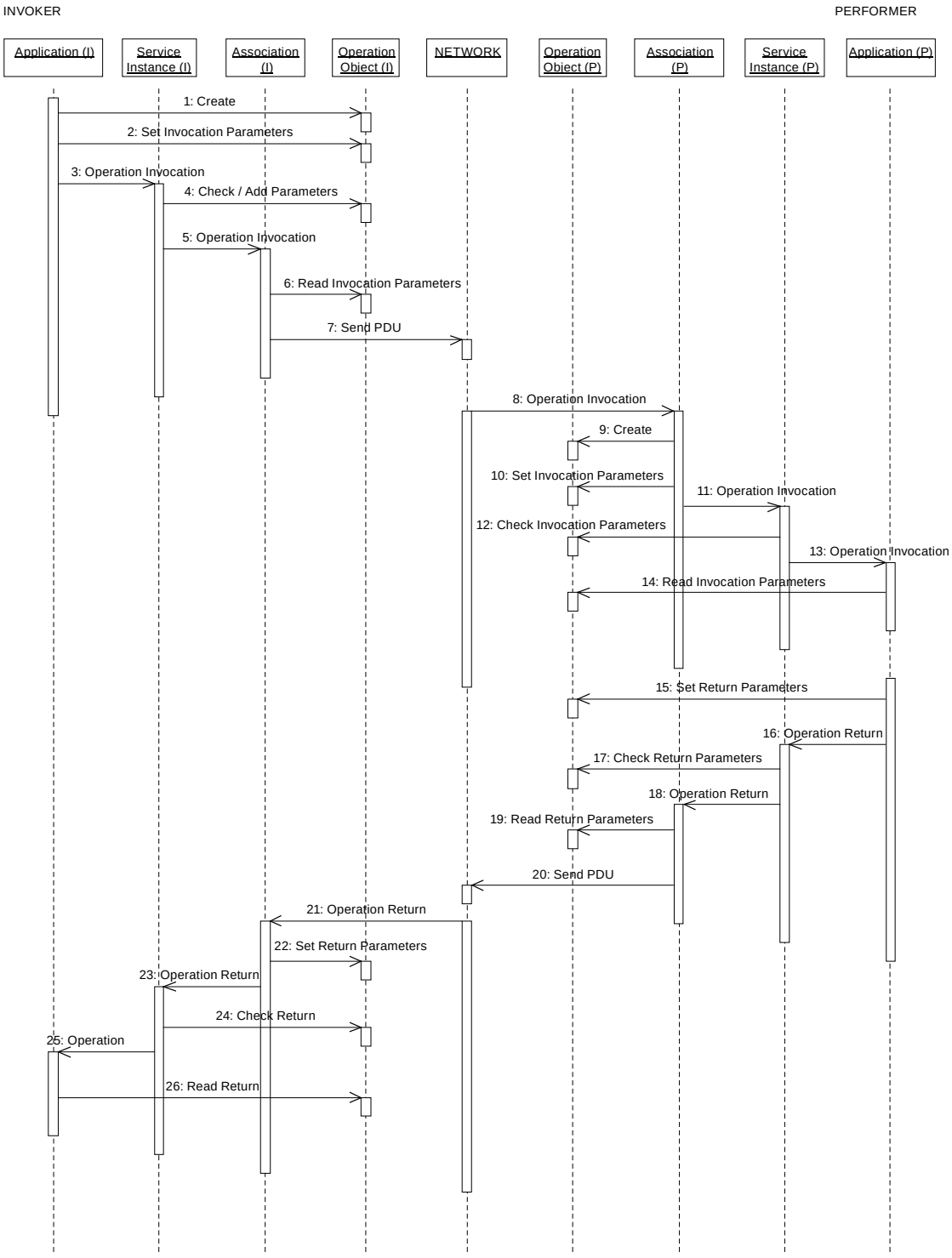

Figure F-6: Sequence Diagram for Use of Operation Objects

When receiving an invocation PDU, the association on the performer side creates the operation object using the operation object factory. It decodes the PDU, passes the parameters to the operation object and passes a reference to the operation object to the service instance via the interface `ISLE_SrvProxyInform` (steps 8-9-10-11).

The service instance verifies the validity of the invocation in its current state, checks the parameters, places the reference on a list of pending local returns and passes the reference to the application via the interface `ISLE_ServiceInform` (steps 12-13). For certain operations, the service instance performs the operation itself, but in this scenario it is assumed that the operation is performed by the application.

The application reads the invocation parameters from the operation object, performs the operation, stores the return parameters to the operation object, and passes the reference to the operation object to the service instance via the interface `ISLE_ServiceInitiate` (steps 14-15-16).

The service instance verifies the validity of the return in its current state, checks the parameters, removes the reference from the list of pending local returns and passes the reference to the association (steps 17-18).

The association on the performer side reads the return parameters from the operation object, encodes the data, constructs the PDU, and transmits it to the peer proxy on the invoker side (steps 19-20).

On the invoker side the association decodes the PDU, locates the operation object holding the invocation by means of the invocation identifier, and stores the return parameters to that object. It removes the object from the list of pending returns and passes the reference to the service instance via the interface `ISLE_SrvProxyInform` (steps 21-22-23).

The service instance verifies the validity of the return in its current state, checks the parameters, cancels the return timer, removes the reference from the list of pending remote returns, and passes the reference to the application (steps 24-25).

The application can now read the return parameters from the operation object (step 26).

To ensure proper memory management, the methods `AddRef()` and `Release()` must be called on the interface of the operation object as described in the following for the invoker and the performer side. Reference counting and the methods `AddRef()` and `Release()` are explained in annex D.

F3.2.2 Invoker Side

a) When the application creates the operation object in step 1 the reference count is set to one by the factory.

b) The service instance calls `AddRef()` when receiving the reference in step 3, setting the reference count to 2.

c) The association calls `AddRef()` when receiving the reference in step 5, setting the count to 3.

d) The association calls `Release()` after performing step 23, setting the count to 2.

e) The service element calls `Release()` after performing step 25, setting the count to 1.

f) The application calls `Release()` when it no longer needs the operation object. This call sets the count to zero and automatically deletes the operation object.

F3.2.3 Performer Side

a) When the association creates the operation object in step 9 the reference count is set to one by the factory.

b) The service instance calls `AddRef()` when receiving the reference in step 11, setting the reference count to 2.

c) The application calls `AddRef()` when receiving the reference in step 13, setting the count to 3.

d) The application calls `Release()` after performing step 16, setting the count to 2.

e) The service element calls `Release()` after performing step 28, setting the count to 1.

f) The association calls `Release()` after performing step 19. This call sets the count to zero and automatically deletes the operation object.

F3.3 USER SIDE BINDING

Figure F-7 shows a scenario in which an SLE user application creates a service instance and then binds the service instance with the provider. For processing of the service element, the scenario shows one specific implementation option, where the service instance creates an association as part of the BIND operation. An implementation might also create the association when the service instance is created. The scenario does not include error cases.

The application creates the service instance using the interface `ISLE_SIFactory` exported by the API Service Element (step 1). Before the service instance can be bound, it must be configured, passing it the parameters set by service management (step 2). When all parameters have been set the application informs the service instance, which checks the configuration (step 3).

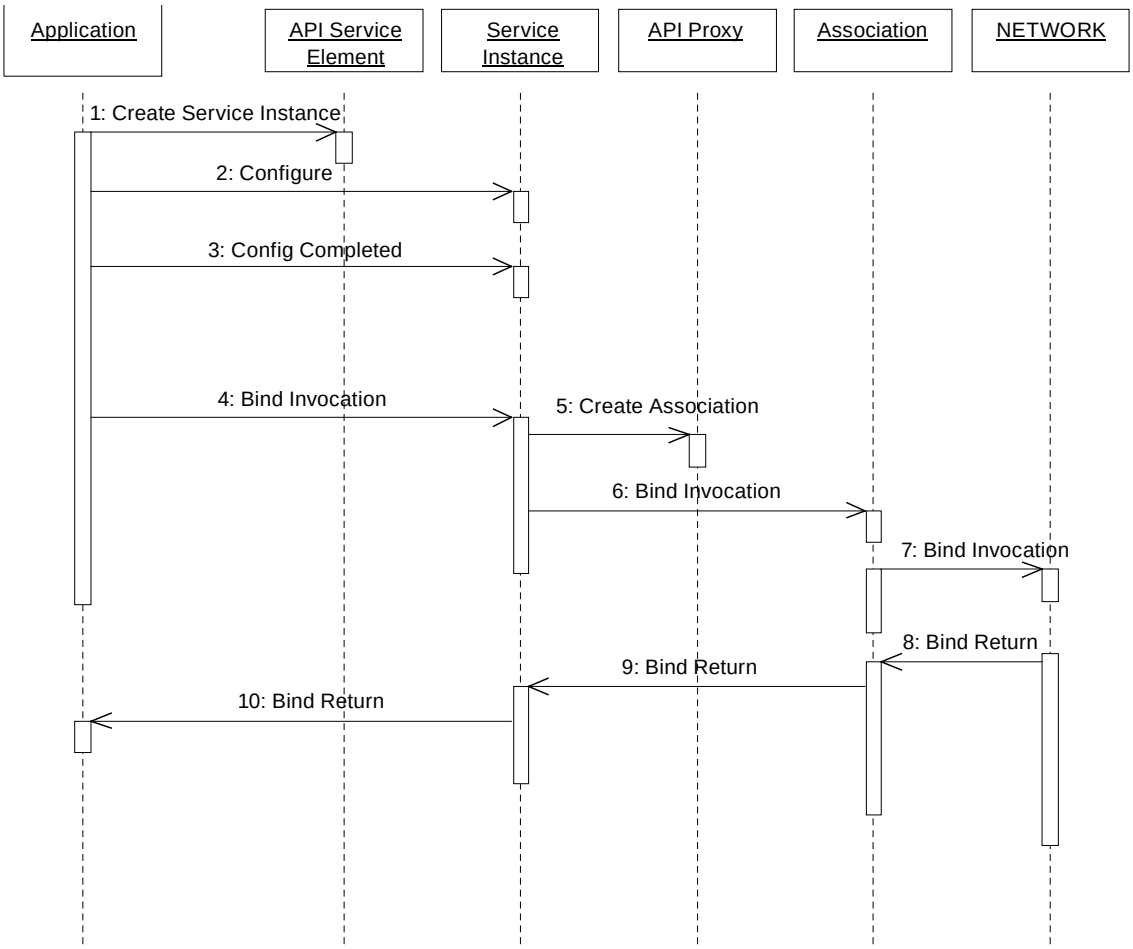

Figure F-7: User Side Binding (User Initiated Bind)

When the service instance has been configured it can be bound. The application requests binding by passing a BIND invocation via the interface `ISLE_ServiceInitiate` (step 4). The service instance creates a new association using the interface `ISLE_AssocFactory` exported by the API Proxy and forwards the BIND invocation to the association via the interface `ISLE_SrvProxyInitiate` (steps 5-6). The association establishes a connection to the provider, encodes the BIND invocation arguments, and transmits the BIND invocation PDU (step 7). As indicated in the diagram, this step is not necessarily performed within the thread of control in which the association received the BIND invocation. The association might also transmit the BIND invocation PDU as part of the connection establishment procedure.

When the BIND return arrives, it is passed through the layers of the API as described for standard operation processing in F3.2 (steps 8-9-10). The state of the association and of the service instance are set to ‘bound’.

F3.4 USER SIDE UNBINDING

The scenario shown in figure F-8 describes processing of the UNBIND operation on the user side for user-initiated binding. For processing of the service element, the scenario assumes that the service instance releases the association as part of the UNBIND operation. An implementation might also retain the association and use it again if the service instance is re-bound. The scenario also assumes that the application does not intend to re-bind, but releases the service instance after unbinding. Component internal activities are indicative only and are displayed in parentheses to highlight this fact. Error cases are not covered.

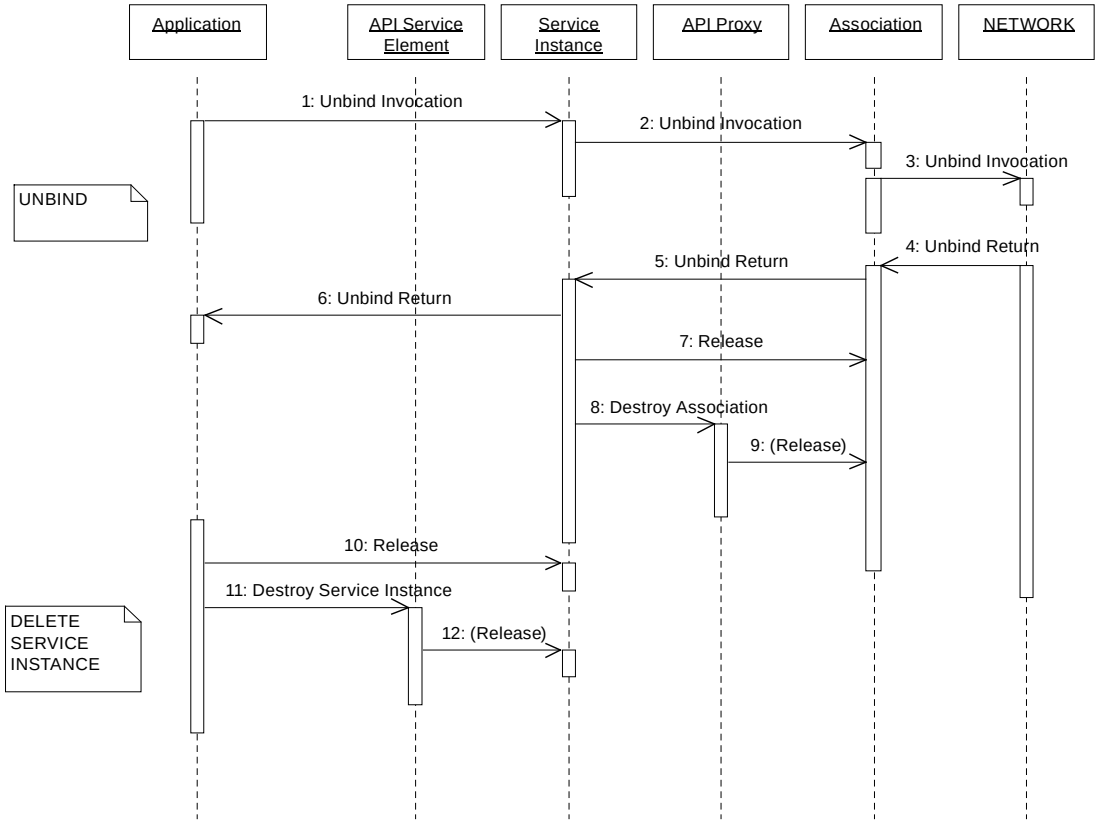

Figure F-8: User Side Unbinding (User Initiated Bind)

The UNBIND invocation is passed through the layers of the API as described for standard processing of operations in F3.2 (steps 1-2-3). As indicated in the diagram, the association might process step 3 at a later time, e.g., after transmission of PDUs that are queued for transmission. The association might also transmit the BIND invocation PDU as part of the connection release procedure.

Reception of the UNBIND return from the network interface concludes connection release. The association sets its state to ‘unbound’ and forwards the UNBIND return to the service element (steps 4-5).

The service instance sets its state to ‘unbound’ and forwards the return to the application (step 6). It then releases the interface of the association and requests the API Proxy to destroy the association via the interface `ISLE_AssocFactory`, which causes the proxy to release the association object and any resources that might be allocated to it (steps 7-8-9).

To delete the service instance the application releases the interface of the service instance and requests the API Service Element to destroy the service instance via the interface ISLE_SIFactory (steps 10-11). The service element releases the service instance object and any resources that may be allocated to it (step 12).

F3.5 PROVIDER SIDE BINDING

Creation of a service instance and subsequent processing of a BIND invocation by an SLE service provider for user-initiated binding is shown in figure F-9. Component internal activities are indicative only and are displayed in parentheses to highlight this fact. Error cases are not covered.

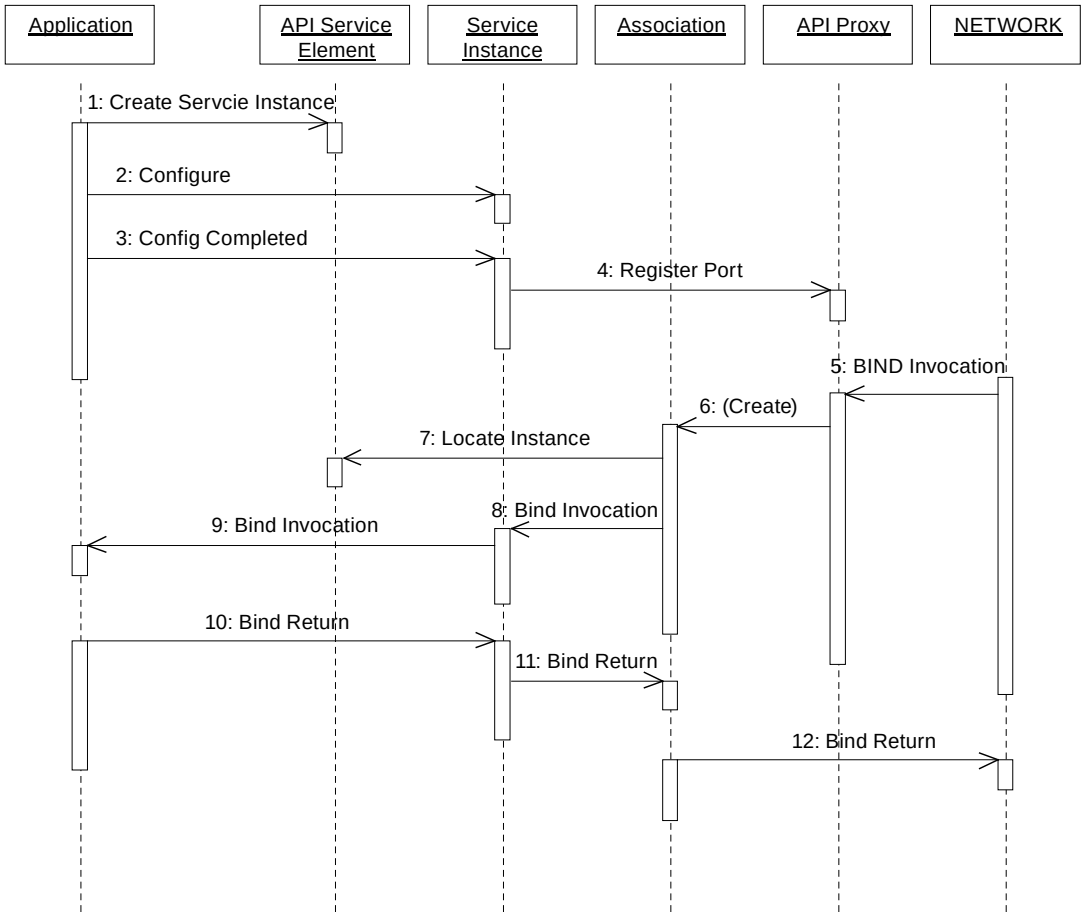

Figure F-9: Provider Side Binding (User Initiated Bind)

The application creates the service instance using the interface `ISLE_SIFactory` exported by the API Service Element (step 1). Then the application configures the service instance, passing it the parameters set by service management (step 2). When all parameters have been set the application informs the service instance, which checks the configuration and registers the port with the API Proxy (steps 3-4).

When receiving a BIND invocation from the network interface, the API Proxy creates a new association for the service type requested in the PDU and passes the BIND invocation to that object (steps 5-6). The association performs initial checks and then informs the API Service Element via the interface `ISLE_Locator` passing it the BIND invocation PDU and a reference to its interface `ISLE_SrvProxyInitiate` (step 7).

If the service element can locate the service instance and has successfully performed all checks it returns a reference to the interface `ISLE_SrvProxyInform` of the service instance. The association passes the BIND invocation to the service instance using that interface (step 8). The service instance finally forwards the BIND invocation to the application (step 9).

The BIND return is passed through the layers of the API as described for standard operation processing in F3.2 (steps 10-11-12). For a BIND return with a positive result the service instance sets its state to ‘ready’; the association completes the connection establishment procedure and sets its state to ‘bound’.

F3.6 PROVIDER SIDE UNBINDING

Figure F-10 shows a scenario where a service instance on a provider system is being unbound (assuming user-initiated binding) and subsequently deleted by the application, because the ‘unbind reason’ is set to ‘end’. Component internal activities are indicative only and are displayed in parentheses to highlight this fact. Error cases are not covered.

An UNBIND invocation received from the network interface is passed through the layers of the API as described for standard operation processing in F3.2 (steps 1-2-3). The association and the service instance both set their state to ‘unbind pending’.

The application eventually issues the UNBIND return (step 4). The service instance sets its state to ‘unbound’ and forwards the return to the association. Subsequently it releases the interface of the association to which it holds a reference (steps 5-6). The association completes the connection release procedure and informs the API Proxy that the association has completed (steps 7-8). The API Proxy releases the association object and all resources that might be allocated to the association (step 9).

The application deletes the service instance when it is informed that the scheduled provision period has ended (step 10). Note that the API issues this notification also when the unbind reason is set to ‘end’.

To delete the service instance, the application releases the interface to which it holds a reference and then instructs the API Service Element to destroy the service instance via the interface `ISLE_SIFactory` (steps 11-12).

The service element instructs the service instance to de-register its port from the API Proxy and then releases the service instance object and any resources that may be allocated to it (steps 13-14-15).

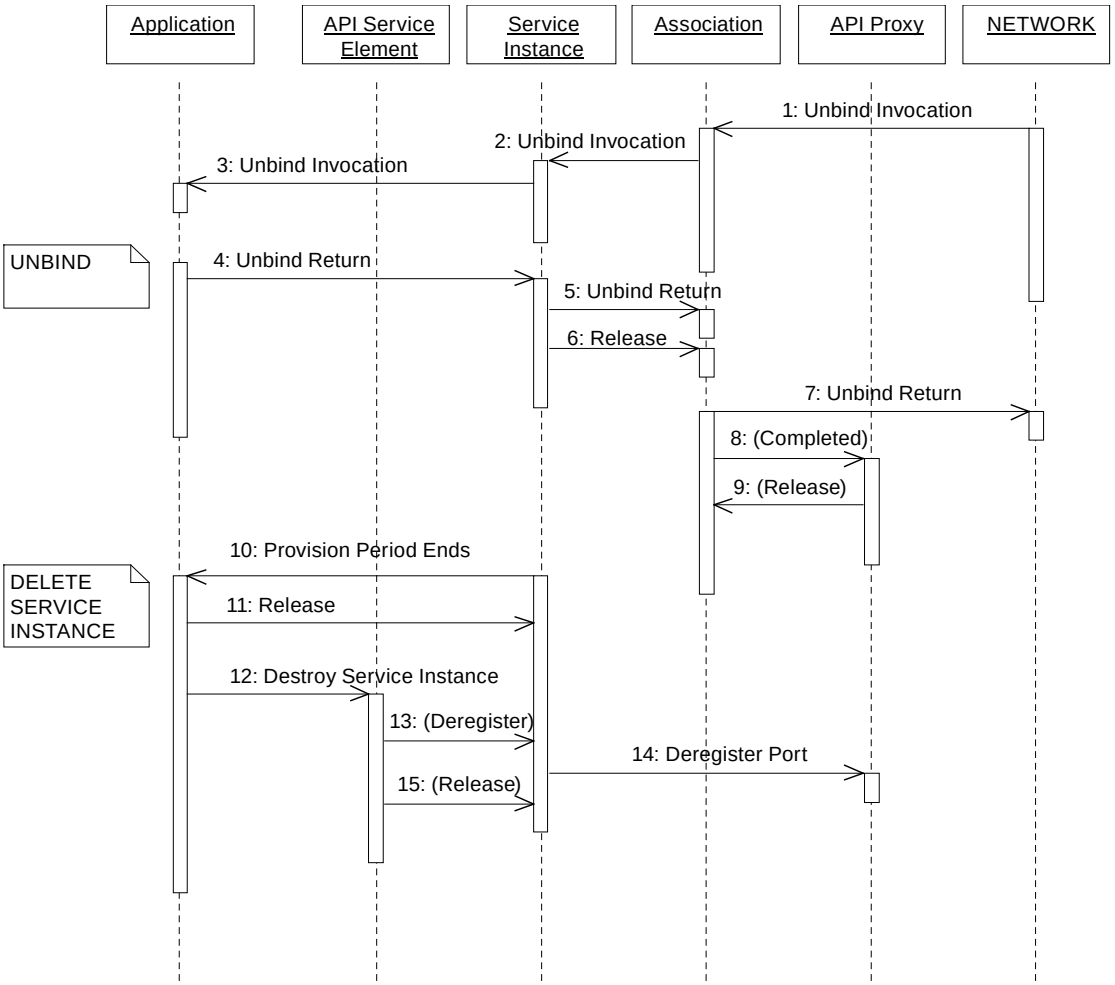

Figure F-10: Provider Side Unbinding (User Initiated Bind)

ANNEX G

INTERFACE CROSS REFERENCE

(Informative)

G1 INTRODUCTION

This annex provides a cross-reference between interfaces by listing, for each of the API components and for the SLE Application:

a) the interfaces exported by the component (or the application);

b) the potential clients of the interface;

c) he interfaces required by the component (or the application);

d) the methods that can be used to obtain a reference to the interface;

e) the potential suppliers of the interface.

It is stressed that an implementation of a component should not make any assumption about the client of an exported interface or the supplier of a required interface. This information is supplied for the only purpose to verify completeness and consistency of the API specification.

Interfaces of individual operation objects exported by the component SLE Operations, and interfaces of individual utility objects exported by the component SLE Utilities, are not specifically considered. Because handling of the interfaces is identical for all these objects, the tables reference these interfaces by ‘interfaces of operation objects’ and ‘interfaces of utility objects’.

Clients (column marked ‘**Clients**’ in the table header) and suppliers (column marked ‘**S**’ in the table header) of interfaces are abbreviated as:

PX	API Proxy
SE	API Service Element
SO	SLE Operations
SU	SLE Utilities
SA	SLE Application

The applicability of interfaces is indicated in a separate column (marked '**A**' in the table header) as:

Mandatory (M) if the interface must be supported or used by all implementations.

Conditional (C) if the interface is associated with a set of options from which an implementation can select or depends on the service type supported by the component.

Optional (O) if the interface is needed for one of the other options specified in annex E.

An additional column (marked '**N**' in the table header) references one or several notes, which are listed below the table.

The information in this annex has been derived from the specifications in section 3 and annexes A and E. It does not provide any new specifications. In case of any discrepancies between this annex and the specifications in section 3 and annexes A and E the latter shall apply.

G2 API PROXY

G2.1 EXPORTED INTERFACES

Interface	Clients	A	N
`ISLE_ProxyAdmin`	SA and SE	M	1
`ISLE_AssocFactory`	SE	C	2
`ISLE_SrvProxyInitiate`	SE (service instance)	M	3
`ISLE_Concurrent`	SE	C	
`ISLE_Sequential`	SE	C	
`ISLE_TimeoutProcessor`	SA or SE	C	4
`ISLE_EventProcessor`	SA or SE	C	4
`ISLE_TraceControl`	SE	O	5

NOTES

1 The interface is used by the application for configuration of the component and by the service element for dynamic port registration if the service element supports responding associations.

2 The interface is required when the proxy supports associations in the initiator role.

3 The interface must be supported by every association object.

4 The interface is only required in combination with `ISLE_Sequential`. The client is the supplier of the corresponding interface `ISLE_TimerHandler` or `ISLE_EventMonitor`.

5 If tracing is supported, then the interface must be supported by the API Proxy and by every association object.

G2.2 REQUIRED INTERFACES

Interface	Interface reference obtained by	S	A	N
`ISLE_Locator`	input argument to `ISLE_ProxyAdmin::Configure()`.	SE SA	C	1
`ISLE_OperationFactory`	input argument to `ISLE_ProxyAdmin::Configure()`.	SO SA	M	2
`ISLE_UtilFactory`	input argument to `ISLE_ProxyAdmin::Configure()`.	SU SA	M	3
`ISLE_Reporter`	input argument to `ISLE_ProxyAdmin::Configure()`.	SA	M	
`ISLE_SrvProxyInform`	a) input argument to `ISLE_AssocFactory::CreateAssociation()`; b) output argument of `ISLE_Locator::LocateInstance()`.	SE SA	M	4
`ISLE_EventMonitor`	input argument to `ISLE_Sequential::StartSequential()`.	SE SA	C	5
`ISLE_TimerHandler`	input argument to `ISLE_Sequential::StartSequential()`.	SE SA	C	5
`ISLE_Trace`	input argument to `ISLE_TraceControl::StartTrace()`.	SA	O	
interfaces of operation objects	a) input argument of one of the methods in `ISLE_SrvProxyInitiate`; b) call to `ISLE_OperationFactory`.	SE SO	M	6
interfaces of utility objects	a) passed by methods of other interfaces; b) call to `ISLE_UtilFactory`.		M	7

NOTES

1 The interface is passed by the application but is actually supplied by the component API Service Element.

2 The interface is passed by the application but is actually supplied by the component SLE Operations.

3 The interface is passed by the application but is actually supplied by the component SLE Utilities.

4 Method a) applies to associations in the initiator role and method b) to associations in the responder role.

5 The interface is only needed in combination with `ISLE_Sequential`.

6 Operation objects might be passed by the service element; they are actually implemented by the component SLE Operations.

7 Utility objects can be passed to the proxy by various methods; they are implemented by the component SLE Utilities.

G3 API SERVICE ELEMENT

G3.1 EXPORTED INTERFACES

Interface	Clients	A	N
ISLE_SEAdmin	SA	M	
ISLE_SIFactory	SA	M	
ISLE_SIAdmin	SA	M	1
I<SRV>_SIAdmin	SA	C	2
I<SRV>_SIUpdate	SA	C	2
ISLE_SIOpFactory	SA	M	1
ISLE_ServiceInitiate	SA	M	1
ISLE_Locator	SA and PX	C	3
ISLE_SrvProxyInform	PX (association)	M	1
ISLE_Concurrent	SA	C	
ISLE_Sequential	SA	C	
ISLE_TimeoutProcessor	SA	C	4
ISLE_EventProcessor	SA	C	4
ISLE_TraceControl	SA	O	5

NOTES

1 The interface is supported for every service instance.

2 The interface is supported for service instances in the provider role, if the interface is defined for the service type supported by that service instance.

3 The interface must be supported by a service element that uses associations in the responder role. The application requires the interface reference only for configuration of the component API Proxy; it does not use any of the methods in the interface.

4 The interface is only required in combination with `ISLE_Sequential`.

5 The interface must be supported by the API Service Element and by every service instance object.

G3.2 REQUIRED INTERFACES

Interface	Interface reference obtained by	S	A	N
`ISLE_ProxyAdmin`	input argument to `ISLE_SEAdmin::AddProxy()`	PX SA	M	1
`ISLE_OperationFactory`	input argument to `ISLE_SEAdmin::Configure()`	SO SA	M	2
`ISLE_UtilFactory`	input argument to `ISLE_SEAdmin::Configure()`	SU SA	M	3
`ISLE_Reporter`	input argument to `ISLE_SEAdmin::Configure()`	SA	M	
`ISLE_ServiceInform`	input argument to `ISLE_SIFactory::CreateServiceInstance()`	SA	M	
`ISLE_AssocFactory`	call to `ISLE_ProxyAdmin::QueryInterface()`	PX	M	4
`ISLE_SrvProxyInitiate`	a) output argument of `ISLE_AssocFactory::CreateAssociation()` b) input argument to `ISLE_Locator::LocateInstance()`	PX	M	5
`ISLE_EventMonitor`	input argument to `ISLE_Sequential::StartSequential()`	SA	C	8
`ISLE_TimerHandler`	input argument to `ISLE_Sequential::StartSequential()`	SA	C	8
`ISLE_Trace`	input argument to `ISLE_TraceControl::StartTrace()`	SA	O	
`ISLE_Concurrent`	call to `ISLE_ProxyAdmin::QueryInterface()`	PX	C	4 6
`ISLE_Sequential`	call to `ISLE_ProxyAdmin::QueryInterface()`	PX	C	4 6
`ISLE_TraceControl`	a) call to `ISLE_ProxyAdmin::QueryInterface()` b) call to `ISLE_SrvProxyInitiate::QueryInterface()`	PX	O	4 7
interfaces of operation objects	a) input argument of one of the methods in `ISLE_SrvProxyInform`; b) input argument to `ISLE_Locator::LocateInstance()`; c) input argument of one of the methods in `ISLE_ServiceInitiate`; d) call to `ISLE_OperationFactory`.	PX SA SO	M	9
interfaces of utility objects	a) passed by methods of other interfaces; b) call to `ISLE_UtilFactory`.		M	10

NOTES

1 The interface is passed by the application but is actually supplied by the component API Proxy.

2 The interface is passed by the application but is actually supplied by the component SLE Operations.

3 The interface is passed by the application but is actually supplied by the component SLE Utilities.

4 The interface can also be obtained by a call to `QueryInterface()` on any other interface exported by the same component object.

5 Method a) is used for associations in the initiator role; method b) applies to associations in the responder role.

6 The interface is needed for control of the proxies linked to the service element.

7 The interface is needed to forward trace requests to the proxies linked to the service element. An interface is required for the API Proxy (method a)) and for every association to support tracing of individual service instances (method b)).

8 The interface is only needed in combination with `ISLE_Sequential`.

9 Operation objects might be passed by the application or the proxy; they are actually implemented by the component SLE Operations.

10 Utility objects can be passed to the service element by various methods; they are implemented by the component SLE Utilities.

G4 SLE OPERATIONS

G4.1 EXPORTED INTERFACES

Interface	Clients	A	N
`ISLE_OperationFactory`	SE and PX	M	
interfaces of operation objects	SA, SE and PX	M	

G4.2 REQUIRED INTERFACES

Interface	Interface reference obtained by	S	A	N
`ISLE_UtilFactory`	input argument to the bootstrap creator function `<impl-id>_CreateOperationFactory()`	SA SU	M	1
`ISLE_Reporter`	optional input argument to the bootstrap creator function `<impl-id>_CreateOperationFactory`	SA	O	2
interfaces of utility objects	call to `ISLE_UtilFactory`	SU	M	

NOTES

1 The interface is passed by the application but is actually supplied by the component SLE Utilities.

2 Supply of the interface by the application is optional; i.e., the corresponding argument can be set to NULL. If supplied, implementation may use the interface to report errors and inconsistencies in the attributes of operation objects.

G5 SLE UTILITIES

G5.1 EXPORTED INTERFACES

Interface	Clients	A	N
`ISLE_UtilFactory`	SA, SE, PX and SO	M	
interfaces of utility objects	SA, SE, PX and SO	M	

G5.2 REQUIRED INTERFACES

Interface	Interface reference obtained by	S	A	N
`ISLE_TimeSource`	input argument to the bootstrap creator function `<impl-id>_CreateUtilFactory()`	SA	O	

G6 SLE APPLICATION

G6.1 EXPORTED INTERFACES

Interface	Clients	A	N
`ISLE_ServiceInform`	SE	M	
`ISLE_Reporter`	SE and PX	M	
`ISLE_TimerHandler`	SE and PX	C	1
`ISLE_EventMonitor`	SE and PX	C	1
`ISLE_Trace`	SE and PX	O	2
`ISLE_TimeSource`	SU	O	

NOTES

1 The interface is required only in combination with `ISLE_Sequential`. It is passed to the API Proxy via the API Service Element, if the interface between these two components is also sequential.

2 The interface is passed to the API Proxy via the API Service Element.

G6.2 REQUIRED INTERFACES

Interface	Interface reference obtained by	S	A	N
ISLE_ProxyAdmin	call to the bootstrap creator function <impl-id>_CreateProxy().	PX	M	1
ISLE_SEAdmin	call to the bootstrap creator function <impl-id>_CreateServiceElement().	SE	M	1
ISLE_Locator	call to ISLE_SEAdmin::QueryInterface().	SE	C	2 3
ISLE_OperationFactory	call to the bootstrap creator function <impl-id>_CreateOperationFactory().	SO	M	1 4
ISLE_UtilFactory	call to the bootstrap creator function <impl-id>_CreateUtilFactory().	SU	M	1
ISLE_SIFactory	call to ISLE_SEAdmin::QueryInterface().	SE	M	3
ISLE_SIAdmin	call to ISLE_SIFactory::CreateServiceInstance().	SE	M	5
I<SRV>_SIAdmin	call to ISLE_SIAdmin::QueryInterface().	SE	C	6
I<SRV>_SIUpdate	call to ISLE_SIAdmin::QueryInterface().	SE	C	6
ISLE_SIOpFactory	call to ISLE_SIAdmin::QueryInterface().	SE	M	6
ISLE_ServiceInitiate	call to ISLE_SIAdmin::QueryInterface().	SE	M	6
ISLE_Concurrent	call to ISLE_SEAdmin::QueryInterface().	SE	C	3
ISLE_Sequential	call to ISLE_SEAdmin::QueryInterface().	SE	C	3
ISLE_TimeoutProcessor	input argument to ISLE_TimerHandler::StartTimer().	SE	C	7
ISLE_EventProcessor	input argument to ISLE_EventMonitor::AddEvent().	SE	C	7
ISLE_TraceControl	a) call to ISLE_SEAdmin::QueryInterface(); b) call to ISLE_SIAdmin::QueryInterface().	SE	O	3 8
interfaces of operation objects	a) input argument of one of the methods in ISLE_ServiceInitiate; b) call to ISLE_SIOpFactory::CreateOperation().	SE SO	M	9
interfaces of utility objects	a) passed by methods of other interfaces; b) call to ISLE_UtilFactory.		M	10

NOTES

1 The application could also request the creator function to return a different interface (e.g., `IUnknown`) and then use `QueryInterface()` to retrieve this interface.

2 The application needs this interface only for configuration of the API Proxy; it does not use any of its methods.

3 `QueryInterface()` can also be called on any other interface exported by the same component object.

4 The application needs this interface for configuration of the API Proxy and the API Service Element. Applications should use the interface `ISLE_SIOpFactory` to create operation objects.

5 The application could also request the method `CreateServiceInstance()` to return any other interface of a service instance and obtain `ISLE_SIAdmin` by calling `QueryInterface()` on that interface.

6 The interface could also be obtained by a call to `QueryInterface()` on any other interface of the service instance or by the call to `ISLE_SIOpFactory` (instead of requesting `ISLE_SIAdmin`).

7 The interface is only required in combination with `ISLE_Sequential`.

8 Method a) is used to control traces for the complete API; method b) provides an interface to control tracing for an individual service instance.

9 An application obtains operation objects from the service instance in the API Service Element; they are actually implemented by the component SLE Operations.

10 Utility objects can be passed to the service element by various methods; they are implemented by the component SLE Utilities.

ANNEX H

INDEX TO DEFINITIONS

(Informative)

This annex provides an index to the terminology defined in the references.

Term	Reference
(data) type	reference [15]
(data) value	reference [15]
abstract binding	reference [3]
abstract object	reference [3]
abstract port	reference [3]
abstract service	reference [3]
Abstract Syntax Notation One (ASN.1)	reference [15]
active (state)	4
association	references [4], [5], [6], [7], and [8]
bound (state)	4
client	1.6.1.4.5
communications service	references [4], [5], [6], [7], and [8]
complete (online delivery mode)	references [4], [5] and [6]
component	1.6.1.4.2
confirmed operation	references [4], [5], [6], [7], and [8]
flow control	2.3.3.4.2.2 and 2.3.3.4.2.3
initiator	reference [19]
interface	1.6.1.4.6
invocation	references [4], [5], [6], [7], and [8]
invoker	reference [3]
latency limit	references [4], [5] and [6]
object identifier	reference [15]

Term	Reference
offline delivery mode	reference [3]
offline frame buffer	references [4], [5] and [6]
online delivery mode	reference [3]
online frame buffer	references [4], [5] and [6]
operation	reference [3]
parameter	references [4], [5], [6], [7], and [8]
performance	references [4], [5], [6], [7], and [8]
performer	reference [3]
physical channel	reference [3]
port identifier	references [4], [5], [6], [7], and [8]
provider-initiated	references [4], [5] and [6]
relative distinguished name (RDN)	reference [17]
release timer	references [4], [5] and [6]
responder	reference [19]
return (of an operation)	references [4], [5], [6], [7], and [8]
return data	reference [3]
service agreement	reference [3]
service instance provision period	references [4], [5], [6], [7], and [8]
service provider (provider)	reference [3]
service user (user)	reference [3]
SLE Complex	reference [3]
SLE Complex Management	reference [3]
SLE data channel	reference [3]
SLE functional group (SLE-FG)	reference [3]
SLE protocol data unit (SLE-PDU)	reference [3]
SLE service data unit (SLE-SDU)	reference [3]
SLE service package	reference [3]

Term	Reference
SLE System	reference [3]
SLE transfer service instance	reference [3]
SLE transfer service production	reference [3]
SLE transfer service provision	reference [3]
SLE Utilization Management	reference [3]
space link	reference [3]
space link data channel	reference [3]
space link data unit (SL-DU)	reference [3]
space link session	reference [3]
telemetry frame	references [4], [5] and [6]
timely (online delivery mode)	references [4], [5] and [6]
transfer buffer	references [4], [5] and [6]
transfer frame	reference [3]
unbound (state)	4
unconfirmed operation	references [4], [5], [6], [7], and [8]
user-initiated	references [4], [5] and [6]

ANNEX I

ACRONYMS AND ABBREVIATIONS

(Informative)

This annex expands the acronyms used throughout this Recommended Practice.

AIF	Application Interface
API	Application Program Interface
CCSDS	Consultative Committee for Space Data Systems
CIF	Client Interface
CLTU	Command Link Transmission Unit
COM	Component Object Model
FSP	Forward Space Packet
GUID	Globally Unique Identifier
ID	Identifier
IEC	International Electrotechnical Commission
IID	Interface Identifier
ISO	International Organization for Standardization
MIF	Management Interface
NIF	Network Interface
OCL	Object Constraint Language
OMG	Object Management Group
PDU	Protocol Data Unit
PIF	Proxy Interface
RAF	Return All Frames
RCF	Return Channel Frames
ROCF	Return Operational Control Field
SHA	Secure Hash Algorithm
SI	Service Instance
SII	Service Instance Identifier
SLE	Space Link Extension
SRV	Service
UML	Unified Modeling Language
UTC	Coordinated Universal Time

ANNEX J

INFORMATIVE REFERENCES

(Informative)

[J1] *Procedures Manual for the Consultative Committee for Space Data Systems*. CCSDS A00.0-Y-9. Yellow Book. Issue 9. Washington, D.C.: CCSDS, November 2003.

[J2] *Space Communication Cross Support—Service Management—Service Specification*. Draft Recommendation for Space Data System Standards, CCSDS 910.11-R-3. Red Book. Issue 3. Washington, D.C.: CCSDS, October 2008.

[J3] *Space Link Extension—Application Program Interface for Transfer Services—Summary of Concept and Rationale*. Report Concerning Space Data System Standards, CCSDS 914.1-G-1. Green Book. Issue 1. Washington, D.C.: CCSDS, January 2006.

[J4] *Space Link Extension—Application Program Interface for Transfer Services—Application Programmer's Guide*. Report Concerning Space Data System Standards, CCSDS 914.2-G-2. Green Book. Issue 2. Washington, D.C.: CCSDS, October 2008.

[J5] *The COM/DCOM Reference*. COM/DCOM Product Documentation, AX-01. San Francisco: The Open Group, 1999. <http://www.opengroup.org/products/publications/catalog/ax01.htm>

[J6] *Unified Modeling Language (UML)*. Version 1.5, formal/2003-03-01. Needham, MA: Object Management Group, March 2003. <http://www.omg.org/technology/documents/modeling_spec_catalog.htm>

[J7] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* 2nd ed., Reading, MA: Addison-Wesley, 1999.

[J8] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.