The Consultative Committee for Space Data Systems

Report Concerning Space Data System Standards

SPACE LINK EXTENSION— APPLICATION PROGRAM INTERFACE FOR TRANSFER SERVICES— APPLICATION PROGRAMMER'S GUIDE

INFORMATIONAL REPORT

CCSDS 914.2-G-2

GREEN BOOK

October 2008

The Consultative Committee for Space Data Systems

Report Concerning Space Data System Standards

SPACE LINK EXTENSION— APPLICATION PROGRAM INTERFACE FOR TRANSFER SERVICES— APPLICATION PROGRAMMER'S GUIDE

INFORMATIONAL REPORT

CCSDS 914.2-G-2

GREEN BOOK

October 2008

AUTHORITY

Issue:	Informational Report, Issue 2
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 reflects the consensus of technical panel experts from CCSDS Member Agencies. The procedure for review and authorization of CCSDS Reports is detailed in the *Procedures Manual for the Consultative Committee for Space Data Systems*.

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

FOREWORD

This document is a technical **Report** for use in developing ground systems for space missions and has been prepared by the **Consultative Committee for Space Data Systems** (CCSDS). The Application Program Interface described herein is intended for missions that are cross-supported between Agencies of the CCSDS.

This **Report** contains background and explanatory material to supplement the CCSDS Recommended Standards and Recommended Practices defining the SLE Application Program Interface for Transfer Services (references [9], [10], [12], [13], [14], [15], and [16]).

Through the process of normal evolution, it is expected that expansion, deletion, or modification of this document may occur. This Report 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 914.2-G-1	Space Link Extension—Application Program Interface for Transfer Services—Application Programmer's Guide, Informational Report, Issue 1	January 2006	Original issue, superseded
CCSDS 914.2-G-2	Space Link Extension—Application Program Interface for Transfer Services—Application Programmer's Guide, Informational Report, Issue 2	October 2008	Current issue

CONTENTS

CONTENTS (continued)

Section Page

Figure

Table

1 INTRODUCTION

1.1 PURPOSE

This Application Programmer's Guide is a Report that has been prepared specifically for software developers, intending to use the API within their applications.

It provides tutorial material for software developers wishing to integrate the API into SLE user applications or SLE provider applications. In particular, it explains how to create and configure API components and discusses a number of scenarios demonstrating how an application can use the API.

The Application Programmer's Guide provides tutorial material and does not replace the API Recommended Practice documents (references [10], [12], [13], [14], [15] and [16]). It does, however, identify what parts of the API Recommended Practice should be consulted by application programmers.

Because the API Recommended Practice documents provide some freedom to implementers of API components, this Report cannot provide all information application programmers need. Missing information should be provided by the documentation provided by API implementers.

1.2 SCOPE

This Report assumes that the reader is familiar with CCSDS Space Link Extension concepts and has a general understanding of SLE transfer services. Readers not familiar with these topics should read the CCSDS Report *Cross Support Concept – Part 1: Space Link Extension Services* (reference [2]) and the CCSDS Report *Space Link Extension – Application Program Interface for Transfer Services – Summary of Concept and Rationale* (reference [11]) before proceeding with this Report. Knowledge of at least one return link SLE service (e.g., the Return All Frames service, reference [4]) and one forward link SLE service (e.g., the Forward CLTU Service, reference [7]) would help to understand the more detailed information presented in sections 4, 6 and 5 of this Report.

The information contained in this report is not part of the CCSDS Recommended Practice documents for the SLE Application Program Interface for Transfer Services. In the event of any conflict between the specifications in references [9], [10], [12], [13], [14], [15], and [16] and the material presented herein, the former shall prevail.

1.3 DOCUMENT STRUCTURE

1.3.1 ORGANIZATION OF THIS REPORT

This Report is organized as follows:

a) section 1 presents the purpose and scope of this Report and lists the references used throughout the Report;

b) section 2 provides an overview of the SLE API for transfer services;

c) section 3 provides general design information on how a SLE application should be designed, and how the SLE API must be used;

d) section 4 provides general information on how to develop an SLE application using the SLE API (this section is not service specific);

e) section 5 provides return service specific information;

f) section 6 provides forward service specific information;

g) annex A contains a glossary of important terms used throughout this Report;

h) annex B lists the acronyms used in this Report.

1.3.2 SLE SERVICES DOCUMENTATION

The SLE suite of Recommended Standards is based on the cross support model defined in the SLE Reference Model (reference [3]). The services defined by the reference model 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.

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

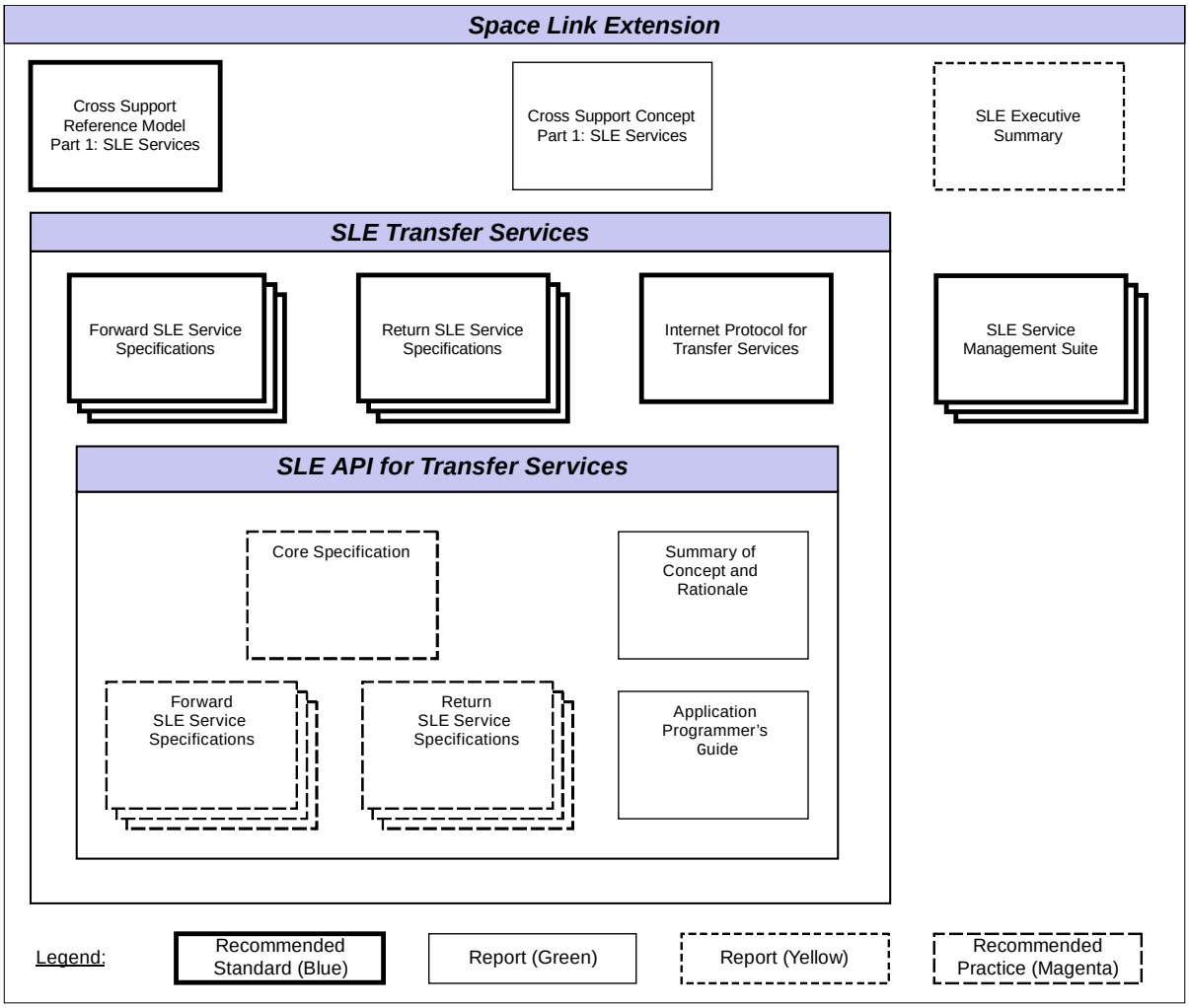

Figure 1-1: SLE Services and SLE API Documentation

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*, a Recommended Practice document specifying the generic part of the API for SLE transfer services.

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 Link 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*, this document.

1.3.3 CONVENTIONS

1.3.3.1 Global Conventions

The names SLE API and API both refer to the Space Link Extension Application Program Interface.

This Report uses the conventions specified in reference [3].

1.3.3.2 Sequence Diagram Conventions

The sequence diagrams presented in this Report show short sequences of operation invocations and returns exchanges. These diagrams follow the UML notation (defined in reference [21]), with the following additional specifications:

a) The focus of control is not displayed.

b) The dashed lines between two SLE API objects depict exchanges of operation invocations and returns via the data communication link. The names of the service operations appear in uppercase and do not include the service specific prefix (SLE, RAF, RCF, ROCF, CLTU, FSP).

c) The lines between the SLE Application and the SLE API objects depict calls of SLE interface methods. A simplified identification of the parameters is provided (only the type of the SLE operation object is provided).

1.3.3.3 Source Code Example Conventions

As typographical convention, source code is presented in monotype font; method name are presented in bold. The following convention apply in the source code example:

a) Only objects for which a reference is got in the code example are released when necessary. Release of other objects should be done following the rules defined in 3.1.4.

b) The C++ comment '`//Error handling code`' indicates that the SLE application should add code to process the error code returned by the SLE API, and should return without executing the code following the error handling code.

c) Each variable starting with '`m_`' is assumed to be a class attribute.

d) In order to simplify the examples, messages are written to the standard output using the standard output stream '`cout`' and the '`<<`' operator. In real SLE application code, appropriate logging or tracing methods should be used.

e) Error codes and enumeration values are written directly to the standard output when necessary, without any conversion to text strings. In real SLE application code, conversion of error codes and enumeration to appropriate text strings should be done.

f) In order to simplify the examples, a downcast is used to convert a reference to an interface from another one when an inheritance relationship between the two interfaces exists and is described in the Recommended Practice documents for SLE transfer services (references [10], [12], [13], [14], [15] and [16]), and when the type of interface can be checked before casting.

1.3.3.4 Typographic Conventions

1.3.3.4.1 Product Name

The SLE API Recommended Practice documents reference product specific interface method names as <product_…>. This notation is used to refer to a specific SLE API implementation. An implementer should refer to related documentation to get detailed information. In this report, the name '`IMP`' (for implementation) is used as product name.

1.3.3.4.2 Operation Names

Names of service operations appear in uppercase and begin with the characters 'SLE-' for SLE service operations, 'RAF-' for RAF service operations, 'RCF-' for RCF service

operations, ‘ROCF-’ for ROCF service operations, ‘CLTU-’ for CLTU service operations, ‘FSP-’ for FSP service operations. In some cases, when the name of the SLE service is of no importance, the service specific suffix ‘SLE-’, ‘RAF-’, ‘RCF-’, ‘ROCF-’, ‘CLTU-’, or ‘FSP-’ is omitted.

1.3.3.4.3 Parameter Names

In the main text, names of parameters of service operations appear in lowercase and are typeset in a fixed-width font (e.g., `forwardToAssoc`).

1.3.3.4.4 Value Names

The values of many parameters discussed in this Report are represented by names. In the main text, those names are shown in single quotation marks (e.g., ‘no such service instance’). The actual value associated with the name is constrained by the type of the parameter taking on that value.

NOTE – The name, i.e., the textual representation, of a value does not imply anything about its type. For example, the value ‘no such service instance’ has the appearance of a character string but might be assigned to a parameter whose type is ‘integer’.

1.3.3.4.5 State Names

SLE service Recommended Standards specify the states of service providers or users. States may be referred to by number (e.g., state 2) or by name. State names are always shown in quotation marks (e.g., ‘active’).

1.4 REFERENCES

The following documents are referenced in this Report. At the time of publication, the editions indicated were valid. All documents are subject to revision, and users of this Report 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.

[1] *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.

[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*. Draft Recommendation for Space Data System Standards, CCSDS 913.1-R-1. Red Book. Issue 1. Washington, D.C.: CCSDS, October 2005.

[10] *Space Link Extension—Application Program Interface for Transfer Services—Core Specification*. Draft Specification Concerning Space Data System Standards, CCSDS 914.0-M-0.1. Draft Recommended Practice. Issue 0.1. Washington, D.C.: CCSDS, October 2005.

[11] *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.

[12] *Space Link Extension—Application Program Interface for Return All Frames Service*. Draft Specification Concerning Space Data System Standards, CCSDS 915.1-M-0.1. Draft Recommended Practice. Issue 0.1. Washington, D.C.: CCSDS, October 2005.

[13] *Space Link Extension—Application Program Interface for Return Channel Frames Service*. Draft Specification Concerning Space Data System Standards, CCSDS 915.2-M-0.1. Draft Recommended Practice. Issue 0.1. Washington, D.C.: CCSDS, October 2005.

[14] *Space Link Extension—Application Program Interface for Return Operational Control Fields Service*. Draft Specification Concerning Space Data System Standards, CCSDS 915.5-M-0.1. Draft Recommended Practice. Issue 0.1. Washington, D.C.: CCSDS, October 2005.

[15] *Space Link Extension—Application Program Interface for the Forward CLTU Service*. Draft Specification Concerning Space Data System Standards, CCSDS 916.1-M-0.1. Draft Recommended Practice. Issue 0.1. Washington, D.C.: CCSDS, October 2005.

[16] *Space Link Extension—Application Program Interface for the Forward Space Packet Service*. Draft Specification Concerning Space Data System Standards, CCSDS 916.3-M-0.1. Draft Recommended Practice. Issue 0.1. Washington, D.C.: CCSDS, October 2005.

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

[18] *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>

[19] Clemens Szyperski, with Dominik Gruntz and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. 2nd ed. Component Software Series. Reading, Massachusetts/New York, New York: Addison-Wesley/ACM Press, 2002.

[20] *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>

[21] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

[22] Don Box. *Essential COM*. The DevelopMentor Series. Reading, MA: Addison-Wesley, 1997.

2 OVERVIEW

2.1 INTRODUCTION

The SLE Reference Model (reference [3]) and the Recommended Standards for SLE transfer services (references [4], [5], [6], [7], [8]) define an abstract model for SLE service production and provision.

The **SLE Application Interface for Transfer Services** (SLE API) offers services for exchange of SLE operation invocations and returns between a SLE service user and a SLE service provider. It specifies interfaces supporting SLE service user applications and SLE service provider applications. The prime objective of the SLE API is to enable development of reusable software packages, which provide a high level, communication technology independent interface to SLE application programs.

A **SLE provider application and a SLE user application** generally comprise the SLE API and many other elements, including other software programs and hardware devices. The API Recommended Practice documents make no assumptions concerning the scope of the application program using the SLE API.

2.2 SLE API CONCEPTS

2.2.1 GENERAL

In order to simplify use of SLE API implementations, the API Recommended Practice documents define an object model, which directly mirror the concepts described in the SLE Reference Model (reference [3]) and the Recommended Standards for SLE transfer services (references [4], [5], [6], [7], and [8]). Application programmers, who are familiar with the SLE service concepts, will therefore find the API intuitive and easy to use.

SLE concepts used for the API object model include:

a) transfer service instances;

b) associations between a service user and a service provider; and

c) transfer service operations.

The following subsections provide a brief introduction to these concepts, which are also described in reference [2].

2.2.2 SERVICE INSTANCES

2.2.2.1 General

The API Service Element provides interfaces for SLE applications to create service instance objects of a specified SLE service type and to configure these objects using the specification of the service instance supplied by management.

All types of service instance objects provide the same interface to invoke SLE operations, which the peer system shall perform. The API transmits these invocations to the performer and delivers the operation return to the application, if the operation is confirmed. Service instance objects deliver operation invocations received from the peer system to the application and provide interfaces for the application to transmit the return for confirmed operations.

2.2.2.2 User Service Instances

SLE user applications create user service instance objects as needed. After creation and configuration, the SLE application invokes the BIND operation on the service instance. The API will then attempt to establish an association to the service provider and report the result back to the application. If the BIND operation succeeds, the state of the service instance is set to 'bound' and the application can invoke further operations as needed. The application terminates use of the service by invoking the UNBIND operation on the service instance. It can then either delete the service instance or re-bind it at a later stage.

2.2.2.3 Provider Service Instances

A SLE service provider application is expected to create a provider service instance object before start of the scheduled provisioning period. Within the provision period, the API accepts a BIND invocation for a service instance, verifies that the request is legal according to the protocol and consistent with the configuration of the service instance, and forwards it to the application. If the BIND invocation is accepted by the application, the API completes the association establishment procedure and sets the state of the service instance to 'bound'. SLE operations are now exchanged until the association is terminated by the UNBIND operation or because of an abort. At the end of the scheduled provisioning period, the application is expected to delete the service instance.

Provider service instances perform SLE operations related to service parameter access and status reporting autonomously, offloading the application from these tasks.

2.2.3 ASSOCIATION OBJECTS

Association objects in the SLE API are responsible for establishment of a data communication association between the SLE service user and the SLE service provider and

for transfer of SLE protocol data units. Conceptually, the association object is created as part of the BIND operation and deleted after completion of the UNBIND operation.

During periods in which a SLE service user and a SLE service provider communicate, a service instance object is linked with exactly one association. Association objects are not visible to SLE applications; they are used exclusively through service instance objects provided by the API Service Element.

2.2.4 SLE OPERATIONS

Transfer services are defined in terms of operations that are requested and performed in the context of a scheduled transfer service instance. Operations are either invoked by the service user and performed by the service provider, or are invoked by the provider and performed by the user. SLE operations can be confirmed, i.e., the result of the operation is returned to the invoker, or unconfirmed.

SLE operations are invoked by sending an invocation PDU from the invoker to the performer. Confirmed SLE operations are terminated by transmission of a return PDU from the performer to the invoker. Transfer of the invocation PDU and transfer of the return PDU are independent actions. A return PDU is associated with an invocation PDU by an invocation identifier.

In the SLE API, these concepts are modeled by SLE operation objects. SLE operation objects provide storage for all parameters defined for a specific SLE operation and provide implementation independent access for reading and writing of these parameters.

On the invoker side, an operation object is initially created by the application, which fills in the invocation parameters of the operation. The operation object is then passed to the SLE API, which reads the parameters, constructs the PDU as required by the technology and transmits it to the peer.

On the performer side, the receiving SLE API creates the operation object and fills in the invocation parameters extracted from the PDU. The operation object is then passed up to the application, which reads the parameters from the object and performs the operation.

2.3 SLE APPLICATION

A SLE application program interacts with the SLE API through a set of interfaces; these interfaces, defined in the Recommended Practice documents for SLE transfer services (references [10], [12], [13], [14], [15] and [16]) allow the SLE application to do the following:

a) Initialize the API by creating the SLE components. One instance of each component must be created via the special ‘factory interfaces’ defined in the API. Detailed information is provided in 4.2.

b) Configure and link the SLE components together. The SLE configuration is provided to the SLE components via the SLE configuration database files. Configuration is detailed in 4.2.

c) Start operation of the SLE components. At startup time the SLE application must indicate which type of behavior will be used, as explained in 4.3.

d) Manage SLE service instances via the SLE service element. This includes creation, deletion, configuration of service instances, as detailed in 4.6.

e) Build, send and receive SLE operation invocations and returns. Details on general handling of SLE operations such as operation creation, configuration, sending and reception are provided in 4.7.

f) Stop operation of the SLE components, as detailed in 4.4.

g) Shut down the API in order to release all resources held by the API and delete the API components, as detailed in 4.5.

SLE application programs must implement a small set of interfaces, by which the SLE API can:

a) pass operation invocations and returns received from the peer system;

b) notify the application of specific events, such as breakdown of the data communications connection; and

c) pass log messages for entry in the system log.

A detailed list of all the interfaces that an SLE application must implement is provided in table 3-1.

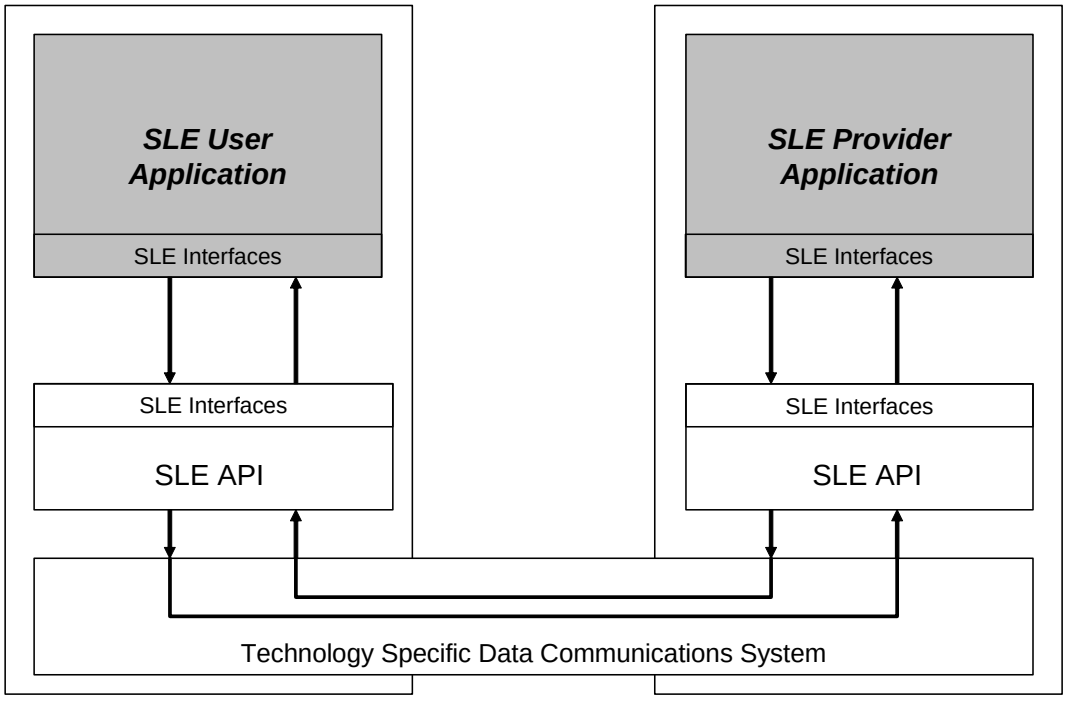

Figure 2-1: SLE Applications

2.4 SLE API COMPONENTS

In order to simplify integration and deployment of SLE implementations, the architecture of the SLE API follows a component based design approach. This approach enables delivery and integration of binary API components instead of source code and allows integration of API components from different sources.

A SLE API component (as defined in reference [10]) is a software module, providing a well-defined service via a set of interfaces.

The SLE API is divided into four SLE components shown in figure 2-2:

a) The component API Service Element implements functionality related to service instance provisioning, which can be clearly separated from service production. This component is responsible for configuration, initialization, and management of the other SLE components.

b) The component API Proxy implements the technology specific features. It implements the data communication system, access control on a system level, and authentication of the peer identity. SLE application does not use this component directly.

c) The component SLE Operations implements the operation objects. SLE applications must use this component to create and handle SLE operations.

d) The component SLE Utilities provides a small set of utility objects, e.g., for memory management, for handling of CCSDS time codes or of SLE service instance identifiers.

Usage of the SLE components by SLE applications is further explained in the following subsections. In particular, 4.2 explains how to create SLE components, configure and link them together.

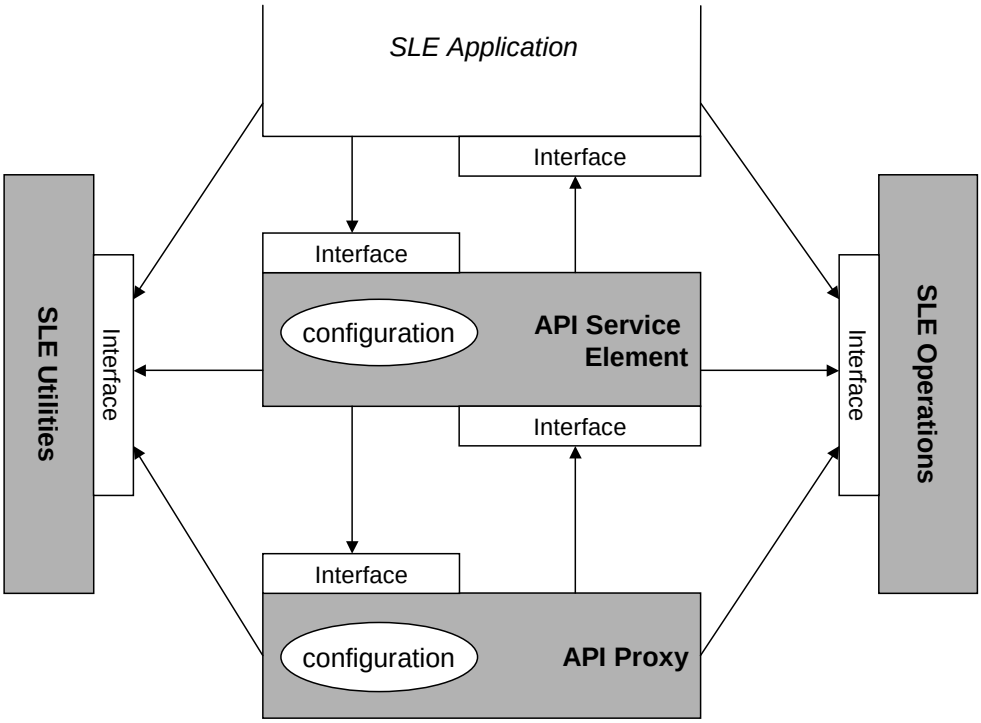

Figure 2-2: SLE API Components

The SLE Application is not a SLE API component, but the client of the SLE API. The services of API components are accessible only via the interfaces defined in the API Recommended Practice documents.

3 GENERAL DESIGN CONSIDERATIONS

3.1 SIMPLE COMPONENT MODEL

3.1.1 INTRODUCTION

The SLE API is based on the concept of integrating independently developed components with the SLE application. For this purpose, the API Core Specification (reference [10]) defines a basic component model, the Simple Component Model (SCM).

The Simple Component Model adopts a limited set of design patterns and conventions from the 'Component Object Model' (COM, see reference [18]). The conventions adopted are restricted to object interactions within the same address space and exclude detection and dynamic loading of components at runtime.

The conventions and design patterns adopted from COM are as follows:

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) Memory allocation and de-allocation of data structure passed from one component to another must follow the memory management rules.

In addition to these conventions, the API Recommended Practice documents also adopt the scheme for definition of result codes from COM.

SLE API components developed according the Simple Component Model do not conform to COM. However, it is possible to develop a SLE application and use SLE API components in a COM environment.

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;

f) programmatic and interactive customization of components.

All interfaces of the SLE API components are defined in the scope of a single address space, which implies that an instance of the API is constrained to one process. An instance of the SLE API is able to handle several service instances concurrently within one process.

NOTE – Applications might want to use a separate process for every service instance or for groups of service instances. Such configurations require that protocol data units for a specific service instance be routed to the correct process. Availability of such features depends on the capabilities of the data communications service, the operating system, and the implementation of the component API Proxy. More information on distribution aspects can be found in reference [11].

3.1.2 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.

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.

The following code shows how the IID for the interface `IRAF_SIUpdate` can be defined. This code is extracted from an include file provided by the SLE API implementation.

```
// Example of the definition of a IID
//-----------------------------------
#define IID_IRAF_SIUpdate_DEF { 0x2f5aeb26, 0x7c28, 0x11d3, \
          { 0x9f, 0x15, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }
```

3.1.3 MULTIPLE INTERFACES OF AN OBJECT

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.

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.

The navigation rules for objects providing multiple interfaces apply to the SLE API interfaces, but also to the interfaces provided by the application to the API.

The SLE API Core Specification (reference [10]) provides interface cross-references for the SLE application interfaces, and details how the SLE application can obtain the requested interfaces.

The following code example shows how an `ISLE_TraceControl` interface can be retrieved from an `ISLE_SEAdmin` interface in order to start the trace on the service element. `pTraceIF` is a reference to a trace interface implemented by the application starting the trace.

```
ISLE_TraceControl*  pIsleTraceControl    = 0;
GUID                guidIsleTraceControl = IID_ISLE_TraceControl_DEF;

//get the trace control interface from the ISLE_SEAdmin interface
//---------------------------------------------------------------
HRESULT eResult = m_pIsleSEAdmin->QueryInterface(guidIsleTraceControl,
                                      (void**)&pIsleTraceControl);
if ( ! FAILED(eResult) )
{
  //start the trace
  eResult = pIsleTraceControl->StartTrace(pTraceIF, sleTL_medium, true);

  //release the trace control interface
  pIsleTraceControl->Release();
  pIsleTraceControl = 0;

  if ( FAILED(eResult) )
  {
    //start trace failed
    //Error handling code
  }
}
else
{
  //Cannot get the trace control interface
  //Error handling code
}
```

NOTE – `FAILED` is a macro defined in the header file, which checks whether the result code reports an error (result codes are detailed in 3.1.6).

The last code example in this section illustrates an implementation of the `QueryInterface()` method for a class implementing the `IUnknown` and `ISLE_ServiceInform` interfaces. In order to allow for easy comparison of GUIDs, it is convenient to use a class wrapper for the GUID Structure. A declaration of such a class with the minimum of methods needed[1] could look like this:

```
class CGuid
{
  public:
    CGuid(const GUID& iid);
    virtual ~CGuid();
    bool operator == (const GUID& iid) const;
    bool operator != (const GUID& iid) const;
  private:
    GUID m_guid;
};
```

GUID is the structure declared in the file `SLE_SCMTypes.h` as

```
typedef struct GUID
{
  unsigned long Data1 ;
  unsigned short Data2 ;
  unsigned short Data3 ;
  unsigned char  Data4[8] ;
} GUID ;
```

Implementation of the class `CGuid` is straightforward and is therefore not detailed here. With this class, the method `QueryInterface` can be implemented as follows:

```
HRESULT APP_ServiceInstance::QueryInterface(const GUID& iid, void** ppv)
{
  GUID guidIUnknown          = IID_IUnknown_DEF;
  GUID guidIsleServiceInform = IID_ISLE_ServiceInform_DEF;

  CGuid g(iid);
  *ppv = 0;
  if ( g == guidIUnknown )
    *ppv = (IUnknown*)this;
  else if (g  == guidIsleServiceInform )
    *ppv = (ISLE_ServiceInform*)this;

  if ( *ppv == 0 )
    return E_NOINTERFACE;

  AddRef();
  return S_OK;
}
```

[1] Addition of further methods would of course make use of the class more convenient.

3.1.4 OBJECT LIFETIME AND REFERENCE COUNTING

3.1.4.1 Overview

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, the API Core Specification (reference [10]) 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 following code example shows how the SLE Utilities component must be created by a SLE application. The method `IMP_CreateUtilFactory` creates the SLE Utilities, and returns a reference to the interface of the created object (`m_pIsleUtilFactory`). The GUID of the interface that shall be returned by the `IMP_CreateUtilFactory` method is provided as first parameter (`guidUtilFactory`). More examples on bootstrap 'creator function' usage can be found in 4.2.

```
HRESULT  eResult          = S_OK;
GUID     guidUtilFactory  = IID_ISLE_UtilFactory_DEF;

// Create the utility factory
//---------------------------
eResult = IMP_CreateUtilFactory(guidUtilFactory, pTimeSourceInterface,
             (void**)&m_pIsleUtilFactory);
if ( FAILED(eResult) )
  //Error handling code
```

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 the reference-count is decremented to zero, `Release()` is allowed to free the object because no one else is using it anywhere. Clients must never invoke the delete operator on interfaces. The rules for reference counting are defined in 3.1.4.2.

In addition, the following conventions apply:

a) `QueryInterface()` implicitly calls `AddRef()` before returning the interface pointer; therefore 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 multithread-safe manner.

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.

3.1.4.2 Reference Counting Rules

The following rules and guidelines are copied from the book 'Essential COM' by Don Box (reference [22]):

a) When a non-null interface pointer is copied from one memory location to another, `AddRef()` should be called to notify the object of the additional reference. `AddRef()` must be called:

   1) when writing a non-null interface pointer to a local variable;

   2) when a callee writes a non-null interface pointer to an out or in-out parameter of a method or function;

   3) when a callee returns a non-null interface pointer as the physical result of a function;

   4) when writing a non-null interface pointer to a data member of an object.

b) `Release()` must be called prior to overwriting a memory location that contains a non-null interface pointer to notify the object that the reference is being destroyed. `Release()` must be called:

   1) prior to overwriting a non-null local variable or data member;

   2) prior to leaving the scope of a non-null local variable;

   3) when a callee overwrites an in-out parameter of a method or function whose initial value is non-null—note that out parameters are assumed to be null on input and must never be released by the callee;

   4) prior to overwriting a non-null data member of an object;

5) prior to leaving the destructor of an object that has a non-null interface pointer as a data member.

c) Redundant calls to `AddRef()` and `Release()` can be optimized away if there is special knowledge about the relationship between two or more memory locations. One common situation in which the special knowledge rule applies arises when passing interface pointers to functions as in-parameters:

– when a caller passes a non-null interface pointer to a function or method through an in-parameter, no call to `AddRef()` or `Release()` is required, as the lifetime of the temporary variable on the call stack is a proper subset of the lifetime of the expression used to initialize the formal argument.

3.1.4.3 Reference Counting Example

The following code example shows how a SLE application could implement the handling of the interface reference to the SLE API service instance in a class (`IMP_SI`) managing a SLE service instance. This class has a private member `m_pIsleSIAdmin` the purpose of which is to store the interface reference to a SLE API service instance. The constructor of the class takes as argument the interface reference to a service instance of the SLE API.

```
IMP_SI::IMP_SI(ISLE_SIAdmin pSIAdmin)
{
  if (pSIAdmin != 0)
  {
    //keep a local reference to the interface
    m_pIsleSIAdmin = pSIAdmin;

    //increment the reference counter
    m_pIsleSIAdmin->AddRef();
  }
}

IMP_SI::~IMP_SI()
{
  if (m_pIsleSIAdmin != 0)
  {
    //release the interface
    m_pIsleSIAdmin->Release();

    //set the local reference to null
    m_pIsleSIAdmin = 0;
  }
}
```

3.1.5 MEMORY MANAGEMENT

In COM there are many interface member functions and APIs which are called by code written by one programming organization and implemented by code written by another. Many of the parameters and return values of these functions are of types that can be passed around by value; however, sometimes there arises the need to pass data structures for which this is not the case. COM defines a universal convention for dealing with these parameters:

a) a memory manager shall be used to allocate and free chunks of memory; and

b) whenever ownership of an allocated chunk of memory is passed through a COM interface, the memory manager must be used to allocate the memory.

Memory management of pointers to interfaces is always provided by member functions in the interface in question. For all the COM interfaces these are the `AddRef()` and `Release()` functions found in the `IUnknown` interface (see 3.1.3). This subsection relates only to non-by-value parameters, which are *not* pointers to interfaces.

`The SLE API provide a memory manager implemented by the SLE Utilities component`. This memory management differs from the COM Specification in the following aspects:

a) only a subset of the methods defined for the interface `IMalloc` is required and supported:

   1) `Alloc()`,

   2) `Realloc()`, and

   3) `Free()`;

b) `the pointer to IMalloc must be obtained by calling the method` CreateIMalloc`() of the Utility Factory.`

3.1.6 RESULT CODES

The API adopts the COM conventions for result codes returned by interface methods. The result codes defined for the SLE API are specified in the API Core Specification (reference [10]).

SLE application must always check return codes returned by the SLE API interface methods. Checking for success or failure must be done using the macros `SUCCEEDED` and `FAILED` defined by COM, and contained in the header file `SLE_RESULT.h` (information on header files is provided in 3.4). In case of failure, SLE application should use the information provided by the result code, as described in the following example.

```
// Create the sync notify operation
//---------------------------------
GUID              guidIrafSyncNotify = IID_IRAF_SyncNotify_DEF;
IRAF_SyncNotify*  pIrafSyncNotify    = 0;

eResult  = pIsleSIOpFactory->CreateOperation (guidIrafSyncNotify,
                                sleOT_syncNotify,
                                (void**)&pIrafSyncNotify);

if ( FAILED(eResult) )
{
  switch (eResult)
  {
    case E_NOINTERFACE:
      cout << "Operation creation failed. "
      cout << "IRAF_SyncNotify interface not supported";
      break;
    case SLE_E_INCONSISTENT:
      cout << "Operation creation failed. "
      cout << "Operation type not supported for this service type";
      break;
    default:
      cout << "Operation creation failed. Failure code: " << eResult;
      break;
  }
  cout << endl;
}
```

For each interface method of the SLE API, the list of possible return codes is described in references [10], [12], [13], [14], [15] and [16].

3.1.7 CONVENTIONS FOR SLE APPLICATIONS

Application programs using the SLE API must adhere to the conventions defined in the previous as clients of SLE API components. In particular, correct functioning of the SLE API can only be ensured if applications handle reference counting correctly.

For reasons of consistency, the interfaces that must be implemented by applications follow the same rules as interfaces provided by the SLE API. This implies that the interface `IUnknown` must be fully supported. However, the API Core Specification (reference [10]) does not define any component object 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 SLE API component still holds a reference to the interface. This requirement can be met by implementing the reference counting scheme as described in 3.1.4. Applications are not required to delete an object when the reference count becomes zero if they use other means to handle object lifetimes.

Every SLE application must provide the interfaces to the SLE API detailed in the next table.

Table 3-1: SLE Application Interfaces

Interface name	Mandatory/Optional	Purpose
`ISLE_ServiceInform`	Mandatory	Provides methods to pass SLE operation invocations and returns, to pass protocol abort, to inform on resumption of data transfer, and to inform on end of the provision period. For each service instance created by the application, a different interface must be provided.
`ISLE_Reporter`	Mandatory	Provides methods for logging and notification of events.
`ISLE_TimerHandler`	Mandatory only for sequential behavior mode	Provides methods to start and stop a timer with a reference to a timeout processor interface.
`ISLE_EventMonitor`	Mandatory only for sequential behavior mode	Provides methods to add and remove event processors in charge of processing external events.
`ISLE_Trace`	Optional	Provides tracing methods.
`ISLE_TimeSource`	Optional	Provides a method for retrieval of the current time.

3.1.8 WORKING IN A COM ENVIRONMENT

The SLE application integrating the SLE API, may be 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 SLE component developed according to the API Core Specification (reference [10]).

3.2 FLOWS OF CONTROL

The API Core Specification (reference [10]) defines two behaviors, a sequential behavior, in which a single flow of control at a time may pass an interface, and a concurrent behavior, in which multiple flows of control can pass an interface concurrently. The SLE application implementers should refer to the specific SLE API product documentation in order to check which behavior is supported. The behavior can be chosen by the application at start-up time.

The selected behavior must be respected by the SLE application. An application intending to use an API implementation supporting concurrent behavior must implement the interfaces provided to the API in a multi-thread safe manner.

For the sequential interface behavior, the API Core Specification (reference [10]) defines interfaces (`ISLE_TimerHandler`, `ISLE_EventMonitor` and `ISLE_EventProcessor`), by which the application offers means for SLE components to wait on external events and to handle timers. The use of these interfaces is further described in 4.3.

3.3 CONFIGURATION

The API Core Specification (reference [10]) specifies a configuration database for the API Proxy and the API Service Element components. These components read the configuration database at start-up of the API on request of the application.

The API Core Specification (reference [10]) does not prescribe the contents of the database. The configuration database might consist of one or more text files or might make use of directory systems or some management database. The implementer of SLE application should refer to specific SLE API implementation documentation to get detailed information.

3.4 SLE API HEADER FILES

The API Core Specification (reference [10]) defines header files that contain interface declarations and type definitions. These files 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. SLE application implementers can also refer to specific SLE API implementation documentations to check whether the header files are in the list of deliverables.

4 DEVELOPING AN SLE APPLICATION

4.1 INTRODUCTION

This subsection describes how SLE applications should use the SLE API. For each aspect, some hints and code examples are provided. Only the common SLE part is described in this subsection. Specific return and forward SLE service aspects are described in sections 5 and 6.

SLE applications should follow the following steps:

a) initialize and configure the SLE API;

b) start the SLE API;

c) create and configure service instances;

d) process SLE operations;

e) delete service instances;

f) stop the SLE API; and

g) delete the SLE API.

Typical scenarios describing operation of the SLE API on the user and provider side are described in 4.12.

4.2 INITIALIZING AND CONFIGURING THE SLE API

4.2.1 INITIALIZING THE SLE API

To be able to use the SLE API, the SLE application must first create the SLE components of the API: the utility factory, the operation factory, the service element, and the proxy component. This creation is done through the creator functions provided by the SLE API:

a) `IMP_CreateUtilFactory()` creates the SLE utilities component;

b) `IMP_CreateOpFactory()` creates the SLE operations component;

c) `IMP_CreateServiceElement()` creates the SLE service element component; and

d) `IMP_CreateProxy()` creates the SLE proxy component.

All these creator functions return to the SLE application a reference to an interface. These references must be stored by the application, and released at the end of the processing. The type of interface returned by the creator function depends on the interface identifier (IID) presented to the creator function.

NOTE – The API Core Specification (reference [10]) offers the possibility for the SLE application to create and manage several proxy components. This Report only considers the case where a single proxy is needed and created by the application.

4.2.2 CONFIGURING THE SLE API

After creation, the SLE API components must be configured before usage. For this purpose, the API Core Specification (reference [10]) defines a configuration database that must be provided by the application to the service element and proxy components. The description of the content of the configuration database is not provided since this content depends on specific SLE API implementation. Implementers shall refer to SLE API product documentation.

4.2.3 LINKING THE SLE COMPONENTS

The linking of the SLE components is done during the creation and the configuration phase:

a) The operation component is linked to the utilities component during creation. When creating the operation factory, reference to the utility factory must be provided.

b) The service element and proxy components are linked to the utilities and operations components during configuration. When configuring the service element and proxy components, reference to the utility and operation factories must be provided.

c) The link between the service element and the proxy is done in two steps:

   1) The `ISLE_Locator` locator interface of the service element permits linking the proxy component to the service element component. This interface, which is only needed for linking, must not be used by the application.

   2) The `AddProxy()` method of the service element administrative interface informs the service element about which proxies have been created and shall be managed.

4.2.4 EXAMPLES

The following example details how the SLE components can be created and linked together by the SLE application.

When creating the utility factory, the SLE application can provide the reference to an external time source interface (`ISLE_TimeSource`) as shown in the example. If the application does not provide this time source reference (setting the time source method parameter to null), the SLE API then uses its internal time source. Handling of time source is further described in 4.9.

When creating the operation factory, the SLE application must provide the reference to a reporter interface (`ISLE_Reporter`) as shown in the example (`pReporterInterface`). This interface is used by the SLE API to report logs and alarms (see 4.10).

```
HRESULT eResult              = S_OK;
//Interface identifiers for the requested interfaces
GUID    guidUtilFactory      = IID_ISLE_UtilFactory_DEF;
GUID    guidMemoryManager    = IID_IMalloc_DEF;
GUID    guidOperationFactory = IID_ISLE_OperationFactory_DEF;
GUID    guidSEAdmin          = IID_ISLE_SEAdmin_DEF;
GUID    guidProxyAdmin       = IID_ISLE_ProxyAdmin_DEF;

// Create the utility component, and request the
// utility factory interface
//-----------------------------------------------
eResult = IMP_CreateUtilFactory(guidUtilFactory, pTimeSourceInterface,
            (void**)&m_pIsleUtilFactory);
if ( FAILED(eResult) )
  //Error handling code

// Create the memory manager object for data
// passed across component boundaries
//----------------------------------------
eResult = m_pIsleUtilFactory->CreateMemoryManager(guidMemoryManager,
                                  (void**)&m_pIMalloc);
if ( FAILED(eResult) )
  //Error handling code

// Create the operation component, presenting the operation factory IID
// and link it to the utility component via the utility factory
//-------------------------------------------------------------------
eResult = IMP_CreateOperationFactory(guidOperationFactory,
            m_pIsleUtilFactory, pReporterInterface,
            (void**)&m_pIsleOperationFactory);
if ( FAILED(eResult) )
{
  //release all interfaces
  m_pIMalloc->Release();
  m_pIMalloc = 0;
  m_pIsleUtilFactory->Release();
  m_pIsleUtilFactory = 0;
  //Error handling code
}
```

```
// Create the service element component, presenting the service
// element administrative IID
//-------------------------------------------------------------
eResult = IMP_CreateServiceElement(guidSEAdmin,(void**)&m_pIsleSEAdmin);

if ( FAILED(eResult) )
{
  //release all interfaces
  m_pIMalloc->Release();
  m_pIMalloc = 0;
  m_pIsleUtilFactory->Release();
  m_pIsleUtilFactory = 0;
  m_pIsleOperationFactory ->Release();
  m_pIsleOperationFactory = 0;
  //Error handling code
}

// Create the proxy component, presenting the proxy administrative IID
//---------------------------------------------------------------------
eResult = IMP_CreateProxy(guidProxyAdmin,(void**)&m_pIsleProxyAdmin);

if ( FAILED(eResult) )
{
  //release all interfaces
  m_pIMalloc->Release();
  m_pIMalloc = 0;
  m_pIsleUtilFactory->Release();
  m_pIsleUtilFactory = 0;
  m_pIsleOperationFactory ->Release();
  m_pIsleOperationFactory = 0;
  m_pIsleSEAdmin->Release();
  m_pIsleSEAdmin = 0;
  //Error handling code
}
```

The following example shows how the service element and proxy SLE components must be configured and linked together. Two configuration database files containing the SLE API configuration are used, `pszSeCfgFile` for the service element configuration and `pszProxyCfgFile` for the proxy configuration.

```
// Configure the service element component, and link it
// to the operation and utilities components
//---------------------------------------------------
eResult = m_pIsleSEAdmin->Configure(pszSeCfgFile,
                            m_pIsleOperationFactory, m_pIsleUtilFactory,
                            pReporterInterface);
if ( FAILED(eResult) )
  //Error handling code
```

```
// Get the locator interface from the ISLE_SEAdmin interface
//-----------------------------------------------------------
GUID           guidLocator = IID_ISLE_Locator_DEF;
ISLE_Locator*  pLocatorInterface    = 0;

eResult = m_pIsleSEAdmin->QueryInterface(guidLocator,
                              (void**)&pLocatorInterface);
if ( FAILED(eResult) )
  //Error handling code

// Configure the proxy component, and link it to the service element,
// and to the operation and utilities components
//-----------------------------------------------------------------
eResult = m_pIsleProxyAdmin->Configure(pszProxyConfigFile,
                                pLocatorInterface,
                                m_pIsleOperationFactory,
                                m_pIsleUtilFactory, pReporterInterface);

if ( FAILED(eResult) )
{
  pLocatorInterface->Release();
  pLocatorInterface = 0;
  //Error handling code
}

// Add the proxy to the service element
//-------------------------------------
eResult = m_pIsleSEAdmin->AddProxy("ISP1", sleBR_initiator,
                             m_pIsleProxyAdmin);

pLocatorInterface->Release();
pLocatorInterface = 0;
```

4.3 STARTING THE SLE API

4.3.1 GENERAL

The way of starting the SLE API components operations depends on the type of behavior. Two behaviors are foreseen in the API Core Specification (reference [10]):

a) sequential behavior, in which methods of the interface must be invoked sequentially by different flows of control;

b) concurrent behavior in which methods of an interface may be invoked concurrently by different flows of control.

During the application design phase, the implementer of an SLE application should first refer to SLE API product documentation to find out which behaviors are supported by the SLE API, and must then decide which behavior shall be used by the application.

The following example describes the start of the API for an application with concurrent behavior. The starting of the processing is done through the concurrent interface (`ISLE_Concurrent`) of the service element. When started, the service element forwards

the start processing to its configured proxies. `m_pIsleSEAdmin` is a reference to the SLE API service element administrative interface.

```
// Get the concurrent interface of the service element
// from the ISLE_SEAdmin interface
//-----------------------------------------------------
GUID guidConcurrent = IID_ISLE_Concurrent_DEF;

eResult = m_pIsleSEAdmin->QueryInterface(guidConcurrent,
                                (void**)&pConcurrentInterface);
if ( FAILED(eResult) )
  //Error handling code

// Start the concurrent interface of the service element
//------------------------------------------------------
eResult = pConcurrentInterface->StartConcurrent();

pConcurrentInterface->Release();
pConcurrentInterface = 0;

if ( FAILED(eResult) )
  //Error handling code
```

4.3.2 SEQUENTIAL BEHAVIOR

In order to start the API with sequential behavior, the SLE application must use the sequential interface (`ISLE_Sequential`) instead of the concurrent one. Moreover, the application must implement objects implementing the `ISLE_EventMonitor` and `ISLE_TimerHandler` interfaces, and must provide references to these two interfaces when starting the sequential behavior.

The `ISLE_EventMonitor` interface is used by the SLE API to register event handles (`SLE_EventHandle`) together with event processor interfaces (`ISLE_EventProcessor`) at the application. The API registers the events by calling the `AddEvent()` method. The SLE application must take the registered event handles into account, and detect during processing any events on these event handles. When a registered event occurs, the application must call the method `ProcessEvent()` of the event processor interface that was registered together with the event handle. When the application is no longer able to handle a registered event, it must call the method `MonitorAbort()` on the event processor interface.

NOTE – How event handles are implemented depends on SLE API product. The implementer should refer to related documentation to get detailed information.

The same mechanism applies for the `ISLE_TimerHandle` interface. This interface is used by the SLE API to start timers at the application. The API starts the timer by calling the `StartTimer()` method, providing a reference to a `ISLE_TimeoutProcessor` interface. When the timer elapses, the application must call the `ProcessTimeout()` method of this timeout processor interface. When the application is no longer able to support

a running timer, it must call the method `HandlerAbort()` on the timeout processor interface.

4.4 STOPPING THE SLE API

The SLE API allows SLE application to gracefully terminate the processing before exiting the process. This termination ensures that all memory and network resources are freed, and that all threads terminate in a controlled manner.

To stop operations of the API, the application must call the terminate method of the API service element, which in turn calls the terminate method of all proxies which it has started. The terminate method to call depends on the type of behavior (concurrent or sequential).

Before stopping the processing of the service element, the application must stop and destroy all the service instances of the service element. Otherwise, the API internally aborts active service instance, and destroy them, without any control from the application.

The following example describes how to stop operations of the SLE API for an application supporting concurrent behavior.

```
// Get the concurrent interface of the service element
// from the ISLE_SEAdmin interface
//-----------------------------------------------------
GUID guidConcurrent = IID_ISLE_Concurrent_DEF;

eResult = m_pIsleSEAdmin->QueryInterface(guidConcurrent,
                              (void**)&pConcurrentInterface);
if ( FAILED(eResult) )
  //Error handling code

// Stop the concurrent interface of the service element
//-----------------------------------------------------
eResult = pConcurrentInterface->TerminateConcurrent();

pConcurrentInterface->Release();
pConcurrentInterface = 0;

if ( FAILED(eResult) )
  //Error handling code
```

Stopping operations of the SLE API for an application supporting sequential behavior is equivalent. The only difference is that sequential interface instead of the concurrent one must be used.

4.5 DELETING THE SLE 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. The application must then call the method `ShutDown()` on the service element and all proxy instances, and release them. This shutdown ensures that all API resources are freed.

```
//Release the factories and the memory manager object
//---------------------------------------------------
m_pIMalloc->Release();
m_pIMalloc = 0;
m_pIsleUtilFactory->Release();
m_pIsleUtilFactory = 0;
m_pIsleOperationFactory->Release();
m_pIsleOperationFactory = 0;

// Shutdown the proxy
//---------------------------
eResult = m_pIsleProxyAdmin->Shutdown();
m_pIsleProxyAdmin->Release();
m_pIsleProxyAdmin = 0;

if ( FAILED(eResult) )
  //Error handling code

// Shutdown the service element
//---------------------------
eResult = m_pIsleSEAdmin->Shutdown();
m_pIsleSEAdmin->Release();
m_pIsleSEAdmin = 0;

if ( FAILED(eResult) )
  //Error handling code
```

4.6 SERVICE INSTANCE MANAGEMENT

4.6.1 GENERAL

Once the SLE API processing is successfully started, the SLE application can create and manage service instances. Creation and deletion of service instance must be done through the `ISLE_SIFactory` interface provided by the service element.

The following subsections describe how to create a service instance, configure it, update it, and delete it.

4.6.2 CREATION OF A SERVICE INSTANCE

When creating a service instance, the SLE application must provide:

a) the interface identifier of the interface the `CreateServiceInstance()` method shall return—the other useful interfaces of the created service instance can be retrieved later by navigation between the interfaces (see 3.1.3);

b) the SLE service type;

c) the role of the service instance (user or provider);

d) a pointer to an implementation of the service inform interface `ISLE_ServiceInform`; and

e) a reference to an interface which type corresponds to the interface identifier passed as first parameter.

The service inform interface is used by the SLE API to inform the application on received operations, protocol abort, end of provision period, resumption of data transfer. For each service instance a separate service inform interface must be provided by the application.

In the following example, a RAF user service instance is created. During creation, the interface identifier `IID_ISLE_SIAdmin_DEF` is passed to the API, to indicate that the application expects a service instance administrative interface (`ISLE_SIAdmin`) as result of the creation. The API, if the creation is successful, then returns in the last parameter of the method `CreateServiceInstance()` the reference to this requested interface. pIsleServiceInform is a reference to the service inform interface (`ISLE_ServiceInform`) implemented by the SLE Application. This interface allows passing of operation invocations and returns to the SLE Application for the created service instance.

```
// Get the SI Factory interface of the service element
// from the ISLE_SEAdmin interface
//-------------------------------------------------
GUID            guidIsleSIFactory = IID_ISLE_SIFactory_DEF;
ISLE_SIFactory* pIsleSIFactory    = 0;

eResult = m_pIsleSEAdmin->QueryInterface(guidIsleSIFactory,
                                (void**)&pIsleSIFactory);
if ( FAILED(eResult) )
  //Error handling code
```

```
// Create a RAF user service instance
// m_pIsleSIAdmin is a pointer to ISLE_SIAdmin
//-----------------------------------------
GUID guidIsleSIAdmin = IID_ISLE_SIAdmin_DEF;

eResult = pIsleSIFactory->CreateServiceInstance(guidIsleSIAdmin,
                              sleAI_rtnAllFrames, sleAR_user,
                              pIsleServiceInform,
                              (void**)&m_pIsleSIAdmin);

pIsleSIFactory->Release();
pIsleSIFactory = 0;

if ( FAILED(eResult) )
  //Error handling code
```

The `m_pIsleSIAdmin` reference created in this example can be used to interface, directly or via other service instance interfaces retrieved via interface navigation, to the created service instance.

4.6.3 CONFIGURATION A OF SERVICE INSTANCE

After creation, the service instance must be configured by the application. The configuration of service instances must be done through the service instance administrative interfaces. The `ISLE_SIAdmin` administrative interface must be used for configuring the common part, and the `I<SRV>_SIAdmin` administrative interface must be used for service specific configuration.

On the user side, the SLE application only need to configure a few common parameters of the service instance via the `ISLE_SIAdmin interface:`

a) service instance identifier;

b) peer identifier;

c) role of the service instance (user or provider);

d) responder port identifier; and

e) timeout value in which return operations must arrive for confirmed operations.

On the provider side, in addition to these common parameters, the SLE application needs to configure:

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

b) service specific configuration parameters via the `I<SRV>_SIAdmin` administrative interface. The parameters depend on the service type and are defined in references [12], [13], [14], [15] and [16] respectively.

When the configuration of a service instance is completed, the application must call the `ConfigCompleted()` method on the administrative interface. The API then checks if the

configuration is complete and consistent. A service instance can be used by the application only after a call to `ConfigCompleted()` returns with success.

NOTE – SLE application must not modify configuration parameters 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 effect of calling the `ConfigCompleted()` method is not the same at the user and the provider side:

a) at the user side, the SLE API simply checks the configuration parameters on completeness and consistency, and is then ready to process SLE operations sent by the application (the first operation sent by the application must be a BIND invoke operation); and

b) at the provider side, the SLE API checks the configuration parameters on completeness and consistency, starts the service instance provision period as configured, and starts listening for incoming connection requests on the configured responder port. If any of these steps failed, a negative result code is returned. Otherwise, the SLE API is ready to receive a BIND operation from the user side during the provision period.

The following example describes the configuration of a RAF provider service instance. `m_pIsleSIAdmin` is a reference to the service instance administrative interface of the RAF provider service instance to configure.

```
// Configure the common part of the service instance
//-------------------------------------------------------------------------
// m_pIsleSII is a pointer to a ISLE_SII object
m_pIsleSIAdmin->Put_ServiceInstanceId(m_pIsleSII);

m_pIsleSIAdmin->Set_PeerIdentifier("SIMSAT");

// Set the role to provider
m_pIsleSIAdmin->Set_BindInitiative(sleAR_provider);

m_pIsleSIAdmin->Set_ResponderPortIdentifier("SIMSATPort1");

// Set the return timeout to 60 seconds
m_pIsleSIAdmin->Set_ReturnTimeout(60);

// m_pIsleStartTime and m_pIsleStopTime are pointers to ISLE_Time objects
m_pIsleSIAdmin->Set_ProvisionPeriod(*pIsleStartTime,*pIsleStopTime);
```

```
// Get the RAF specific service admin interface of the service instance
// from the ISLE_SIAdmin interface
//-----------------------------------------------------------------------
GUID          guidIrafSIAdmin = IID_IRAF_SIAdmin_DEF;
IRAF_SIAdmin* pIrafSIAdmin    = 0;

eResult = m_pIsleSIAdmin->QueryInterface(guidIrafSIAdmin,
                                (void**)&pIrafSIAdmin);
if ( FAILED(eResult) )
  //Error handling code

// Configure the RAF service specific part of the service instance
//---------------------------------------------------------------
// Set the delivery mode to online timely
pIrafSIAdmin->Set_DeliveryMode(rafDm_timelyOnline);

// Set the latency limit to 5 seconds
pIrafSIAdmin->Set_LatencyLimit(5);

// Set the transfer buffer size to 100 operations
pIrafSIAdmin->Set_TransferBufferSize(100);

// Set the statuses
pIrafSIAdmin->Set_InitialProductionStatus(rafPs_running);
pIrafSIAdmin->Set_InitialFrameSyncLock(rafLS_outOfLock);
pIrafSIAdmin->Set_InitialCarrierDemodLock(rafLS_outOfLock);
pIrafSIAdmin->Set_InitialSubCarrierDemodLock(rafLS_outOfLock);
pIrafSIAdmin->Set_InitialSymbolSyncLock(rafLS_outOfLock);

// Indicate that configuration is completed
//-----------------------------------------
eResult = m_pIsleSIAdmin->ConfigCompleted();

pIrafSIAdmin->Release();
pIrafSIAdmin = 0;

if ( FAILED(eResult) )
  // Configuration problem – The service instance cannot be used
  //-----------------------------------------------------------
  //Error handling code
```

4.6.4 UPDATE OF A SERVICE INSTANCE

SLE application in the provider role must update API service instance dynamic parameters, depending on the current processing. For this purpose, the interface `I<SRV>_SIUpdate` must be used. This interface is service specific, and its usage is described in 5.1 and 6.2.

4.6.5 DELETION OF A SERVICE INSTANCE

A SLE user application must delete a service instance when it is no longer needed. Deletion of service instance is only possible in ‘unbound’ state. On the provider side, a SLE provider application must delete a service instance at the end of the provision period. To delete a service instance, the SLE application must do the following:

a) Instruct the SLE API service element to releases all references. This is done by calling the method `DestroyServiceInstance()` of the interface `ISLE_SIFactory`. As parameter, the reference to the `IUnknown` interface of the service instance to delete must be provided.

b) Release all its references to the service instance to delete.

NOTES

1 When calling the `DestroyServiceInstance()` of the interface `ISLE_SIFactory,` it is important to provide the reference to the `IUnknown` interface of the service instance to delete, and not any other reference to this service instance. The SLE API may strictly compare the reference provided as parameter with one internally stored and initialized when the service instance was created.

2 The `DestroyServiceInstance()` method does not necessarily delete the service instance object. It tells the SLE API to remove the service instance and to release all the interfaces the SLE API has on this object. This object, as all the other objects handled by the SLE API, is deleted when its reference counter reaches 0.

The following example shows how a service instance is destroyed, and how interfaces are released. `m_pIsleSIAdmin` is a reference to the service instance administrative interface of the service instance to delete.

```
// Get the IUnknown interface of the service instance to destroy
// from the ISLE_SIAdmin interface
//-------------------------------------------------------------
GUID      guidIUnknown = IID_IUnknown_DEF;
IUnknown* pIUnknown    = 0;

eResult = m_pIsleSIAdmin->QueryInterface(guidIUnknown,
                                (void**)&pIUnknown);
if ( FAILED(eResult) )
  //Error handling code

// Destroy the service instance
//-----------------------------
eResult = m_pIsleSIFactory->DestroyServiceInstance(pIUnknown);

pIUnknown->Release();
pIUnknown = 0;

// Release the reference to the destroyed service instance
//--------------------------------------------------------
m_pIsleSIAdmin->Release();
m_pIsleSIAdmin = 0;

if ( FAILED(eResult) )
  //Error handling code
```

4.7 SLE OPERATIONS

4.7.1 CLASSIFICATION OF SLE OPERATIONS

SLE operations can be confirmed, i.e., the result of the operation is returned to the invoker, or unconfirmed. SLE operation processing depends on the application role.

Confirmed operations are always sent by the user application to the provider application. The user application sends the operation invocation; the provider application receives it, process it, and the result of the processing is sent back to the user in the form of an operation return. This mechanism is fully asynchronous.

NOTE – SLE confirmed operations can be invoked by the user or the provider application. But at the time of writing, none of the SLE transfer services specified a confirmed operation which is invoked by the provider. Therefore, a SLE provider will never receive operation returns, and a SLE user will never receive confirmed operation invocations.

The SLE API always stores references to invocation of confirmed operations, in order to associate them with return operations. SLE applications must use the operation object passed by the API during reception of the invoke operation to send the return operation (the API compares the reference to the operation object). The API also uses a return timeout timer. If no operation return is received for a confirmed operation before this timer elapses, the API aborts the association between the user and the provider.

On the provider side, some received confirmed operations are processed directly by the SLE API, without informing the SLE application. This is the case for the SCHEDULE-STATUS-REPORT and GET-PARAMETER operations. The provider API receives the operation, process it, and returns the corresponding return operation without informing the provider application.

Unconfirmed operations can be sent by the user or the provider application. These operations are not confirmed by sending an operation return (the concept of SLE operation return is only defined for confirmed operations).

4.7.2 USAGE OF SLE OPERATIONS

Once a service instance is created and configured, the SLE application is ready to process SLE operations. Operations are either invoked by the service user and performed by the service provider, or are invoked by the provider and performed by the user.

On the **user side**, the application must do the following:

a) First invoke a BIND operation to initialize the SLE API association to the provider, and wait for the reception of the BIND return from the provider. After successful binding, the service instance is in ‘bound’ state.

b) Once in 'bound' state, the user application can invoke further SLE operations, and must process SLE operation invokes and returns received from the provider.

On the **provider side**, the application must do the following:

a) Wait for the reception of the BIND operation invocation from the user side. This operation must be processed, and a BIND return, with positive or negative result, sent back to the user. If the result is positive, the service instance is in 'bound' state.

b) Once in 'bound' state, the provider application can send invoke SLE operations to the user, and must process invoke SLE operations received from the user.

NOTE – A provider service instance accepts SLE operations only during the configured provision period. At the end of the scheduled period, if the service instance is still bound, the SLE API aborts the association with the user and informs the provider application via a call to the `ProvisionPeriodEnds()` of the `ISLE_ServiceInform` interface.

The following figure illustrates the usage of confirmed and unconfirmed operations between a user and provider application.

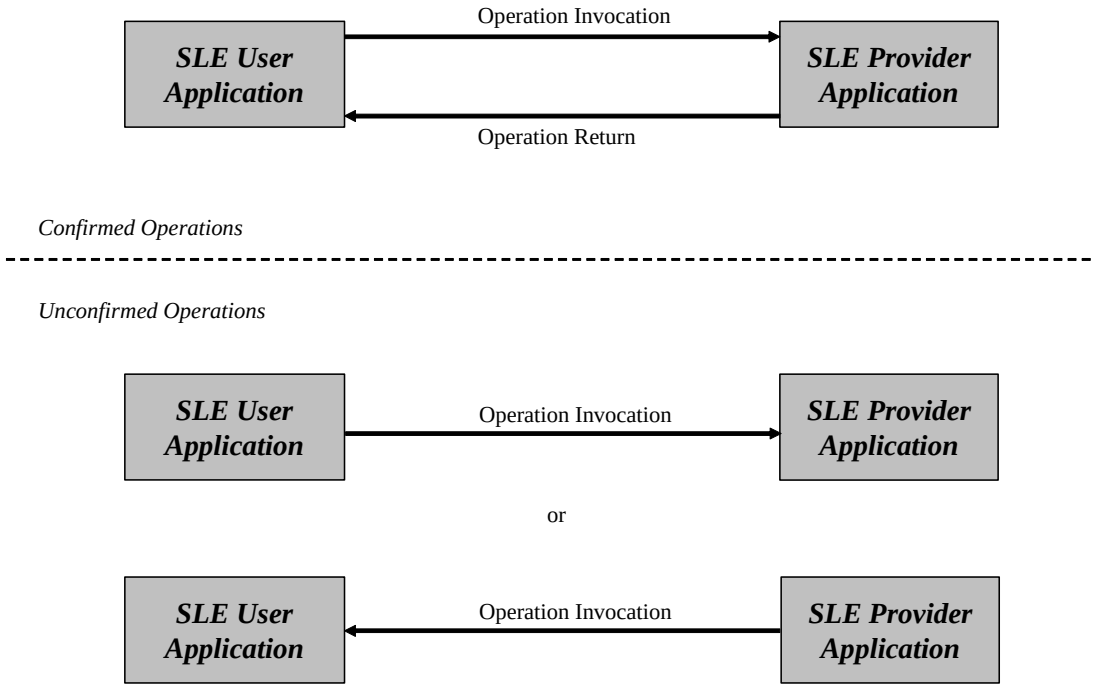

Figure 4-1: SLE Operations Usage Diagram

4.7.3 CREATION AND CONFIGURATION OF SLE OPERATIONS

An SLE application should create SLE operations through the operation factory provided by the service instance. The following are true for this service instance operation factory:

a) It allows creation of operations initially consistent with the service instance. The service instance operation factory already configures the following parameters of each created operation: service type, operation type, confirmed/unconfirmed indication, service instance identifier. Some other operation parameters, depending on the SLE operation type and SLE service type, are also configured. References [10], [12], [13], [14], [15] and [16] specify how operation parameters are configured after creation by the service instance operation factory.

b) It only allows creation of SLE operations which are defined for the service type, and which can be invoked by applications in the defined role.

c) It is available via the `ISLE_SIOpFactory` interface of the service instance.

NOTES

1 The application cannot specify whether the operation is an operation invocation or an operation return at creation time. The interface used to send the operation decides if it is an invocation or a return.

2 The application must never create an operation object and send it as an operation return. An SLE application must use the operation invocation object passed by the API to send the operation return (as explained in 4.7.7).

3 An SLE application could also use the operation factory available through the `ISLE_OperationFactory` interface of the service element. This is not recommended because the service instance operation factory provides a better and safer service.

4 Initialization of operation parameters is specified in references [10], [12], [13], [14], [15] and [16].

The following example shows how a SCHEDULE-STATUS-REPORT operation can be created from the `ISLE_SIOpFactory` interface of the service instance. `m_pIsleSIAdmin` is a reference to the service instance administrative interface of the service instance to which the operation belong.

```
// Get the service instance operation factory interface
// from the ISLE_SIAdmin interface
//-----------------------------------------------------
GUID guidIsleSIOpFactory           = IID_ISLE_SIOpFactory_DEF;
ISLE_SIOpFactory* pIsleSIOpFactory = 0;

eResult = m_pIsleSIAdmin->QueryInterface(guidIsleSIOpFactory,
                                (void**)&pIsleSIOpFactory);
if ( FAILED(eResult) )
  //Error handling code

// Create a schedule status report operation
//------------------------------------------
GUID guidIsleSSR = IID_ISLE_ScheduleStatusReport_DEF;
ISLE_ScheduleStatusReport* pIsleSSR = 0;

eResult = pIsleSIOpFactory->CreateOperation(guidIsleSSR,
                                sleOT_scheduleStatusReport,
                                (void**)& pIsleSSR);

pIsleSIOpFactory->Release();
pIsleSIOpFactory = 0;

if ( FAILED(eResult) )
  //Error handling code
```

Once the SLE operation is created, the application must configure it before passing it to the SLE API. The following example shows the configuration of a SCHEDULE-STATUS-REPORT operation (`pIsleSSR`).

```
// Configure the scheduled status report operation
//------------------------------------------------
// Set the request type to periodically
pIsleSSR->Set_ReportRequestType(sleRRT_periodically);

// Set the reporting cycle to 60 seconds
pIsleSSR->Set_ReportingCycle(60);
```

4.7.4 DELETION OF SLE OPERATIONS

No specific method exists to delete SLE operations. SLE operation objects, as all the other objects handled by the SLE API, are deleted when the reference count reaches 0. When the SLE application do no longer uses an SLE operation, it should simply release it via the `Release()` method.

4.7.5 USING THE MEMORY MANAGER FOR C TYPE DATA PASSED TO OR OBTAINED FROM SLE OPERATION

Certain SLE Operation objects need to be configured using C type data, e.g., TM frame data or TC data. When the application passes these data to the SLE Operation, these data are passed across component borders. Therefore, the API Memory Manager has to be used for allocating these data.

The following example shows how a RAF-TRANSFER DATA operation (`pIrafTFD`) will be configured before passing it to the SLE API.

```
// Allocate the TM frame data
const size_t size = 1000 * sizeof(SLE_Octet);
SLE_Octet *m_TmFrameData = 0;
m_TmFrameData = m_pIMalloc->Alloc(size);

// Fill the data contents from RAF service production
// (to be implemented by application)
readData(m_TmFrameData, size);

// Pass the TM frame data to the RAF transfer data operation
pIrafTFD->Put_Data(m_TmFrameData, size);
m_TmFrameData = 0; // remember, data now belongs to the API
```

When the application obtains these data from the SLE Operation (which is part of the API Operation Factory component), these data are passed across component borders. Therefore, the API Memory Manager has to be used for de-allocating these data after use.

The following example shows how the TM frame data from a RAF-TRANSFER DATA operation can be obtained from the SLE Operation object (`pIrafTFD`) and will be de-allocated after use.

```
// Size and pointer for TM frame data
size_t size = 0;
SLE_Octet *m_TmFrameData = 0;

// Obtain the data and size from the operation
m_TmFrameData = pIrafTFD->Remove_Data(size);

// Release the operation
pIrafTFD->Release();

// Process the data (to be implemented by application)
processData(m_TmFrameData);

// Free the data using the Memory Manager
m_pIMalloc->Free(m_TmFrameData);
```

4.7.6 SENDING SLE OPERATION INVOCATIONS AND RETURNS

To send a SLE operation invocation or return on a service instance, the SLE application must call the method of the service instance initiate interface `InitiateOpInvoke()` for

invocation and `InitiateOpReturn()` for returns. These methods take two parameters, the operation object to send, and a sequence counter. For the sequential behavior the sequence counter is irrelevant and should always be set to zero; its use for interfaces with concurrent behavior is discussed in 4.7.9.

The SLE API checks the parameters of the operation passed as argument of the method `InitiateOpInvoke()`, and checks if the operation fits to the SLE state machine specified in reference [10]. If one check fails, the operation invocation or return is rejected, and an appropriate result code is returned.

The following example shows how a SCHEDULE-STATUS-REPORT invoke operation is sent to the service instance. In this example, the sequence counter is set to 0 since sequential behavior is assumed. `m_pIsleSIAdmin` is a reference to the service instance administrative interface of the service instance on which the operation is sent.

```
// Get the service instance initiate interface
// from the ISLE_SIAdmin interface
//-----------------------------------------
GUID guidIsleServiceInitiate = IID_ISLE_ServiceInitiate_DEF;
ISLE_ServiceInitiate* pIsleServiceInitiate = 0;

eResult = m_pIsleSIAdmin->QueryInterface(guidIsleServiceInitiate,
                                (void**)&pIsleServiceInitiate);
if ( FAILED(eResult) )
  //Error handling code

// Send the invoke operation
//--------------------------
eResult = pIsleServiceInitiate->InitiateOpInvoke(pIsleSSR, 0);

pIsleServiceInitiate->Release();
pIsleServiceInitiate = 0;

if ( FAILED(eResult) )
  //Error handling code
```

4.7.7 RECEIVING SLE OPERATION INVOCATIONS AND RETURNS

The SLE application receives SLE operation invocations and returns from a service instance of the API through the `InformOpInvoke()` and `InformOpReturn()` methods of the `ISLE_ServiceInform` interface. SLE application processing then depends on the application role.

An SLE **user application** must be prepared to receive operation invocations and returns. Only unconfirmed operation invocations and confirmed operation returns can be received. Therefore, the user application never needs to send any operation return in response to operation invocation.

An SLE **provider application** needs only be prepared to receive operation invocations. When receiving a confirmed operation invocation, the application is expected to perform the

operation, store the result (and the diagnostic in case of negative result) to the operation object, and return it to the API. The same operation object that has previously been passed to the application by the service instance must convey the return for the confirmed operation.

NOTE – SLE confirmed operations can be invoked by the user or the provider application. But at the time of writing, none of the SLE transfer services specified a confirmed operation, which is invoked by the provider. Therefore, a SLE provider will never receive operation returns, and a SLE user will never receive confirmed operation invocations.

The SLE API already implements the state tables defined by SLE Recommended Standards for transfer services, and verifies validity of the operation parameters. Therefore, the SLE application does not need to check if the received operation is valid depending on the service instance state, and can minimize the checking of the operation parameters.

As mentioned in 4.7.6 for the method `InitiateOpInvoke()`, the sequence counter passed as argument of the `InformOpInvoke()` method is not relevant for the sequential behavior and can be ignored in that case. Its use for the concurrent behavior is discussed in 4.7.9.

The following example details the implementation of the `InformOpInvoke()` method of a SLE provider application. The example only details processing of BIND and UNBIND operation invocations. The `ProcessBind()` and `ProcessUnbind()` methods are expected to do further processing of the operation. In case of error during the processing, the method `ProcessBind()` returns false and sets the diagnostic to the value that must be returned to the user side. In case of UNBIND no error return is foreseen as a BIND invocation must not be rejected if received in a valid state and state checking has already been performed by the SLE API. In this example, the sequence counter is not used since sequential behavior is assumed.

```
HRESULT APP_RafUserSI::InformOpInvoke (ISLE_Operation* pOperation,
                                       unsigned long nSeqCount)
{
  SLE_OpType                eOpType;
  SLE_ApplicationIdentifier eServiceType;

  // Get the service and operation type of the received invocation
  //--------------------------------------------------------------
  eOpType         = pOperation->Get_OperationType();
  eServiceType    = pOperation->Get_OpServiceType();

  // Log the received operation
  //---------------------------
  cout << "Receive " << eServiceType << eOpType << " invoke operation";

  // Process the operation depending on its type
  //--------------------------------------------
  switch ( eOpType )
  {
```

```
    case sleOT_bind:
    {
      // Downcast to the bind operation
      //-------------------------------
      ISLE_Bind * pIsleBind = (ISLE_Bind *)pOperation;

      // Process the bind operation and set the result and bind diagnostic
      //------------------------------------------------------------------
      SLE_BindDiagnostic bindDiagnostic;
      if (ProcessBind(pIsleBind, bindDiagnostic) == true)
        pIsleBind-> Set_PositiveResult();
      else
        pIsleBind->Set_BindDiagnostic(bindDiagnostic);

      // Send the bind return
      //---------------------
      eResult  = m_pServiceInitiate->InitiateOpReturn(pOperation, 0);
      if ( FAILED(eResult) )
        //Error handling code

      break;
    }

    case sleOT_unbind:
    {
      // Downcast to the unbind operation
      //---------------------------------
      ISLE_Unbind * pIsleUnbind = (ISLE_Unbind *)pOperation;

      // Process the unbind operation and set the result
      // and unbind diagnostic
      //-----------------------------------------------
      SLE_UnbindReason unbindDiagnostic;
      ProcessUnbind(pIsleUnbind)
      pIsleUnbind-> Set_PositiveResult();

      // NOTE: UNBIND must not be rejected if invoked in a valid state
      // and state violations are already dealt with by the API

      // Send the unbind return
      //-----------------------
      eResult = m_pServiceInitiate->InitiateOpReturn(pOperation, 0);
      if ( FAILED(eResult) )
        //Error handling code

      break;
    }

    //handle other operation invocation
    ...
}
```

NOTES

1 In this example, a downcast is used to get a reference to a `ISLE_Bind` operation from the `ISLE_Operation`. This downcast is safe since there is an inheritance relationship between the two interfaces, and since the type of operation is checked before casting.

2 Only code for expected operation type needs to be included into the SLE application, because the SLE API makes sure only operation invocations valid for the service type and application role (user or provider) are passed.

4.7.8 OPERATION RESULT

When a user application receives a confirmed operation return, it must check the operation result. If the operation result is not positive, the application should first get the diagnostic type, and then, depending on this diagnostic type, should get the common or specific diagnostic.

When the SLE API detects an error while processing an operation invocation, it sets the result to negative, sets the appropriate diagnostic, and returns it to the invoker. The negative operation invocation is in most cases not delivered to the SLE application. However, for some specific operation types (defined in references [12], [13], [14], [15] and [16]), the negative operation invocation is forwarded to the provider application for further processing. For example, this is the case for CLTU-TRANSFER-DATA operation (see 6.3). Provider application receiving an operation with negative result should not process it, but simply update some of its parameters if necessary (depending on the service type and operation type), and return it to the user side.

The following example details how a CLTU user application processes a START return operation. The `ProcessStartReturn()` method do further processing of the START return operation. This example can be part of the `InformOpReturn()` method implemented by the SLE application and invoked by the SLE API.

```
switch (pConfirmedOperation->Get_OperationType())
{
  ...

  case sleOT_start:
  {
    // Downcast to the start operation
    //--------------------------------
    ICLTU_Start * pIcltuStart = (ICLTU_Start *) pConfirmedOperation;

    // Check the result of the operation
    //----------------------------------
    if (pIcltuStart->Get_Result() != sleRES_positive)
    {
      // get the diagnostic type
      //------------------------
      switch (pIcltuStart->Get_DiagnosticType ())
      {
        case sleDT_commonDiagnostics:
          cout << "Receive negative START return – Diagnostic="
               << pIcltuStart->Get_Diagnostics();
          break;
```

```
          case sleDT_specificDiagnostics:
            cout << "Receive negative START return – Diagnostic="
                 << pIcltuStart->Get_StartDiagnostic();
            break;

          case sleDT_noDiagnostics:
          default:
            cout << "Receive negative START return – "
                  << "No diagnostic provided";
            break;
        }
      }
      else
      {
        // Process the positive start return
        //----------------------------------
        ProcessStartReturn(pIcltuStart);
      }
      break;
    }
    ...
  }
```

4.7.9 SEQUENCE COUNTING

SLE Recommended Standards require that operation invocations and returns be delivered in the same sequence in which they were sent. This requirement applies to the API in the first place, but can be relevant for application software using the API if this software receives Space Link Data Units (e.g., telemetry frames or telecommands) via some external interface.

If processing is performed within a single thread of control the requirement is met by default, because the data units received from an interface are processed sequentially. In the presence of multiple concurrent threads (concurrent behavior), sequence preservation depends on the specific way in which threads are being used. When different operation invocations and returns passed via the same interface are processed by different threads, then preservation of the original sequence cannot be guaranteed. In order not to constrain the use of multiple threads, the SLE API specification does not require that operation invocations and returns be passed across an interface with concurrent behavior in the original sequence. Instead, the specification requires that the original sequence be identified by a sequence counter such that the receiving party can restore that sequence.

The rules that must be observed by the sending side are:

a) the sequence counter is initially set to zero;

b) the sequence count for a BIND invocation or a BIND return is set to one; and

c) for each subsequent operation, the sequence count is incremented by one.

If the design of the application software ensures that all Space Link Data Units are passed to the method `ISLE_ServiceInitiate::InitiateOpInvoke()` in the original sequence, then it is sufficient to use a local counter, which is incremented every time the

method is invoked. Otherwise, the software must ensure that the sequence counter reflects the original sequence of the data units.

For the receiving side, the SLE API Specification defines the following procedure for reestablishment of the original sequence:

a) the receiving party defines a window in which it accepts sequence counts—the size of this window should be configurable;

b) an operation with a sequence count equal to the next expected sequence count is processed immediately;

c) an operation with sequence count inside the window is stored for later processing;

d) an operation with a sequence count outside the window is rejected with the error code E_SLE_SEQUENCE; and

e) PEER-ABORT invocations are always processed immediately independent of the sequence count.

Obviously, this re-sequencing procedure would not be required if the API Service Element component actually delivers operation invocations and returns to a given interface in a single thread of control. It is strongly recommended, however, not to make any assumptions on how an API component is implemented. As the API specification allows components to deliver data units out of sequence, the option of exchanging API components will only be available if the re-sequencing procedure is actually implemented for the interface `ISLE_ServiceInform`.

4.8 PROTOCOL ABORT

The SLE API proxy component is responsible for monitoring the state of the data communication connection between the user and the provider side. When communication problems are detected, the SLE API aborts the association between the user and the provider and informs the local application using the method `ProtocolAbort()` of the `ISLE_ServiceInform` interface. The diagnostic explaining the reason of the abort is provided as parameter.

4.9 TIME SOURCE

SLE applications have the option of supplying an external time source to the API components. To use this option, the application must provide an implementation for the interface `ISLE_TimeSource` and pass it to the creator function of the component SLE Utilities (see 4.2.1). When the time source interface is supplied by the application, the API components use this interface to retrieve current time. Otherwise, they use the system time.

4.10 LOGGING

The SLE API components generate log messages for important events, and enter them to the system log of the hosting system using the interface `ISLE_Reporter`, passed at configuration time. Each log message is identified by a unique number, which is referenced in the documentation of the specific SLE API product, and is passed to the interface `ISLE_Reporter` when the message is logged.

The `ISLE_Reporter` provides logging and notification facilities:

a) `LogRecord` – This method enters a new log message into the system log. The type of log (`SLE_LogMessageType`) identifies if the log is an alarm or an information message.

b) `Notify` – This method notifies a specific event which requires immediate attention.

Log message identifiers in the range 0 to 999 are reserved for use by the SLE API. These log message identifiers must not be used for messages defined by an SLE application.

SLE applications must implement the `ISLE_Reporter` logging interface, and process the log messages received from the API as appropriate. The logging interface must be provided when creating the operation factory (see 4.2.1), and when configuring the service element and the proxy (see 4.2.2).

SLE Applications can internally use the `ISLE_Reporter` logging interface to issue application logs and to notify events. Such logs should be issued, setting the `SLE_Component` to 'application'.

4.11 TRACING

The SLE API offers tracing facilities, which must be managed through the `ISLE_TraceControl` interface. A SLE application has several possibilities:

a) Start and stop the trace on the API service element. The service element automatically forwards the start of the tracing to all its the service instances (also for further service instance creation). Moreover, if the `forward` argument in the function `StartTrace()` is set to true, the service element forwards the starting of the trace to all its attached proxies.

b) Start and stop the trace only on one API service instance. If the `forward` argument in the function `StartTrace()` is set to true, the service instance forwards the starting of the trace to the association object it uses.

To be able to set the tracing, SLE application must implement the `ISLE_Trace` tracing interface, and must provide mechanism to store and display the trace messages received from the API. When starting the trace, the SLE application must provide the reference to its tracing interface, together with the trace level.

The SLE API supports four trace levels:

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.

c) 'High' – the trace additionally contains a printout of all parameters of the PDU processed.

d) 'Full' – the trace additionally contains a dump of the encoded data sent to and received from the network.

The following example shows how trace is started on a specific service instance. The forward parameter is set to `true`, to indicate that tracing must be forwarded on the association object in use. `m_pIsleSIAdmin` is a reference to the service instance administrative interface of the service instance on which tracing is started.

```
// Get the service trace control interface
// from the ISLE_SIAdmin interface
//----------------------------------------
GUID               guidIsleTraceControl = IID_ISLE_TraceControl_DEF;
ISLE_TraceCOntrol* pIsleTraceControl    = 0;

eResult = m_pIsleSIAdmin->QueryInterface(guidIsleTraceCOntrol,
                                (void**)&pIsleTraceControl);
if ( FAILED(eResult) )
  //Error handling code

// Start tracing
//--------------
bool forwardToAssoc = true;
SLE_TraceLevel level = sleTL_high;
eResult = pIsleTraceControl ->StartTrace(pTrace, level, forwardToAssoc);

pIsleTraceControl->Release();
pIsleTraceControl = 0;

if ( FAILED(eResult) )
  //Error handling code
```

4.12 TYPICAL SCENARIOS FOR SLE APPLICATIONS

4.12.1 GENERAL

SLE application must first initialize, configure, and start the SLE API, as described in 4.2 and 4.3. Once the SLE API is started, it is ready for creation of one or several service instances. It is up to SLE application to decide when service instances should be created and deleted.

The SLE application can decide to create all required service instances after the start of the SLE API, or can create the service instance one after the other, depending on the need. Service instance management is described in 4.6.

After this initialization phase, the SLE application is ready to use service instances for processing of SLE operations. At the end of the processing, the SLE application must orderly stop the API processing. For this purpose, the application:

a) first must terminate the created service instances until they reach the 'unbound' state;

NOTE – For this purpose, it might be necessary on the user side to send a STOP and an UNBIND operation, depending on the current service instance state. On the provider side, the only way for the application to force termination of a service instance is to send a PEER-ABORT operation;

b) destroy all the service instances;

c) stop the SLE API as described in 4.4; and

d) delete the SLE API as described in 4.5.

The following subsections provide scenarios describing how SLE operations should be sent by SLE applications via the SLE API interface methods. Figures presenting sequence diagrams are provided. Sequence diagram notation conventions are described in 1.3.3.2.

4.12.2 BINDING, STARTING, STOPPING, UNBINDING A SERVICE INSTANCE

After a service instance has been created and configured by the SLE user application, it is ready to process SLE operations. On the provider side, the service instance is ready to process SLE operations only during the configured provision period. The first operation sent by the user application must be a BIND operation invocation. When receiving a BIND operation invocation, the provider application must respond with a BIND operation return, with the result (and the diagnostic in case of negative result) set. The same scheme applies to the START, STOP, and UNBIND operations. The following figure shows processing of BIND, START, STOP, and UNBIND operations where all operation results are positive.

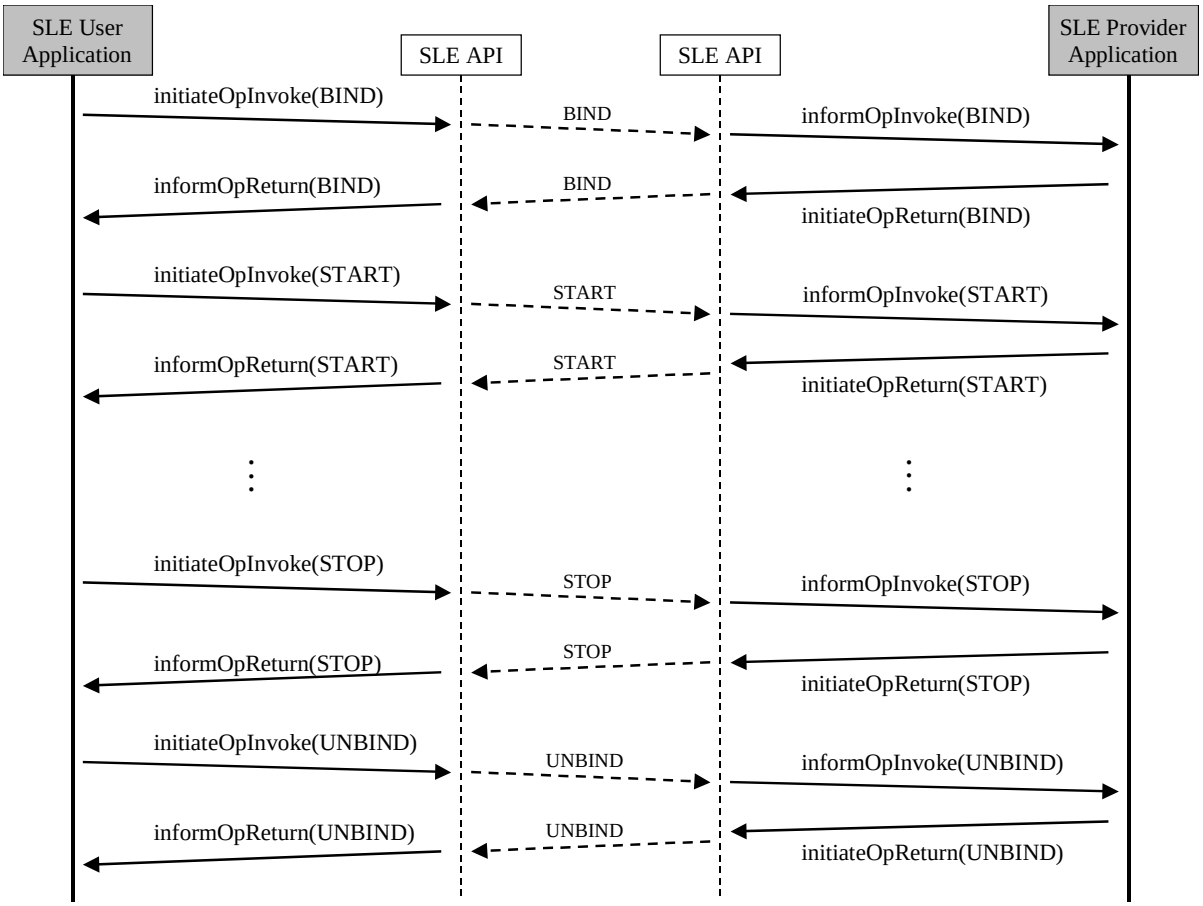

Figure 4-2: Binding, Starting, Stopping, Unbinding Sequence Diagram

NOTES

1 After stopping a service instance, the user application can start it again, by calling the `initiateOpInvoke()` method and passing a START operation.

2 After unbinding a service instance, the user application can bind it again, by calling the `initiateOpInvoke()` method and passing a BIND operation.

3 Following successful processing of a BIND operation, the SLE API is ready to process SCHEDULE-STATUS-REPORT and GET-PARAMETER operations. Also STATUS-REPORT operations might be sent from the provider side.

4.12.3 STATUS REPORTING

After a service instance has been created and configured by the SLE application, and the BIND operation has been performed, the service instance is in 'bound' state. In the 'bound' state, the service instance is able to process SCHEDULE-STATUS-REPORT and STATUS-REPORT operations. The SCHEDULE-STATUS-REPORT operation is always sent by the user application. On the provider side, the operation is processed directly by the SLE API

and is not forwarded to the application. The provider SLE API then sends STATUS-REPORT invocations periodically or only once, depending on the report request type sent in the SCHEDULE-STATUS-REPORT operation. The following figure provides an example where the user application requests a periodic status report with a period of 10 seconds.

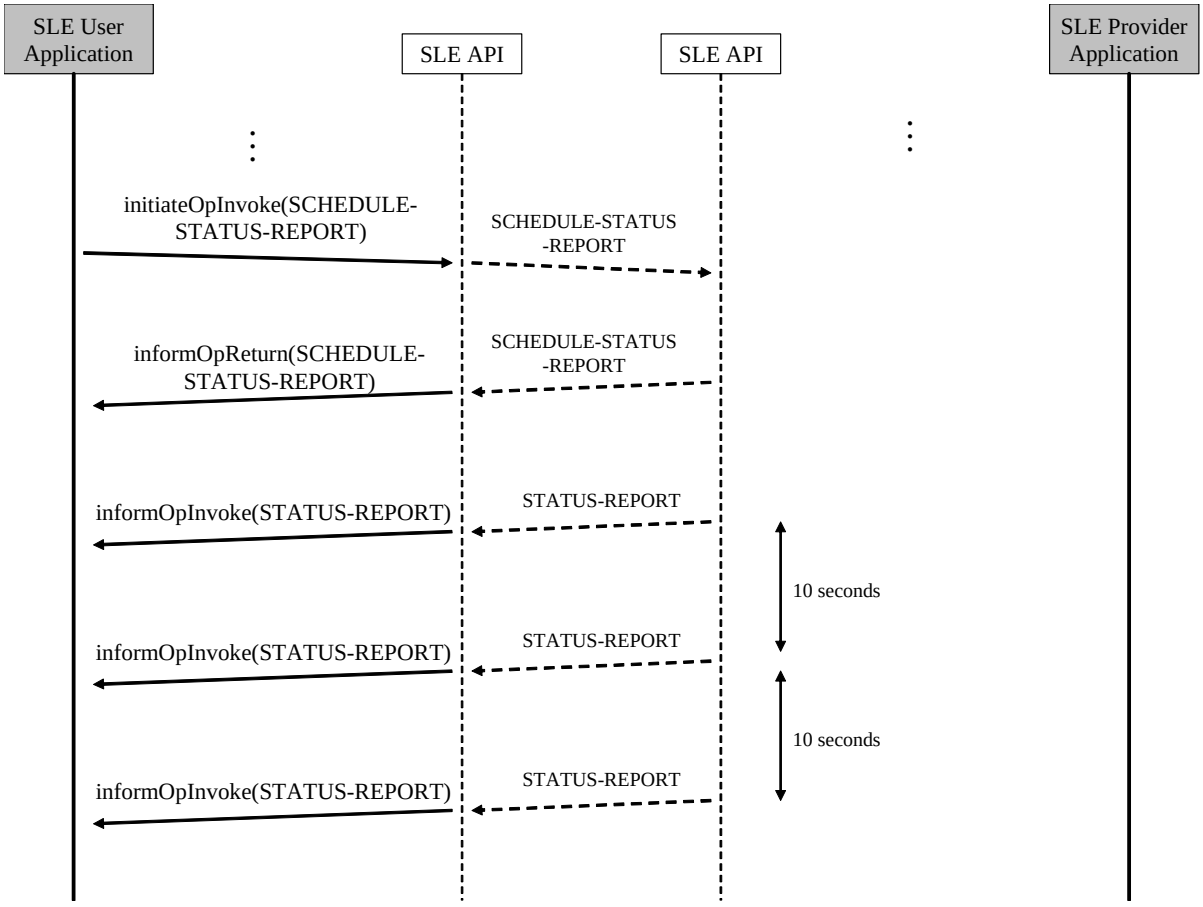

Figure 4-3: Status Reporting Sequence Diagram

NOTES

1 To stop a periodic status report, the SLE user application must send a SCHEDULE-STATUS-REPORT invocation with the request type set to ‘stop’.

2 Status reporting is automatically stopped by the provider service instance of the API when a UNBIND or PEER-ABORT operation is received.

3 Status reporting is used by the SLE user application to get information on the status of the provider service instance. The SLE provider application must update its service instance through the service instance update interface. Update of service instance is detailed in 5.1 and 6.2. This update is not shown in the figure.

An example of creation and configuration of a SCHEDULE-STATUS-REPORT operation is provided in 4.7.3.

The following example shows how a RAF user application processes a received STATUS-REPORT invocation.

```
switch (pOperation->Get_OperationType())
{
  ...

  case sleOT_statusReport:
  {
    // Downcast to the status report operation
    //----------------------------------------
    IRAF_StatusReport *pIrafStatusReport =
                         (IRAF_StatusReport *)pOperation;

    cout << "Receive Status Report from the provider";
    cout << "Number of error free frames delivered is "
         << pIrafStatusReport->Get_NumErrorFreeFrames();
    cout << "Total number of frames delivered is "
         << pIrafStatusReport->Get_NumFrames();
    cout << "Frame synchronisation lock status is "
         << pIrafStatusReport->Get_FrameSyncLock();
    cout << "Carrier demodulation lock status is "
         << pIrafStatusReport->Get_CarrierDemodLock();
    cout << "Subcarrier demodulation lock status is "
         << pIrafStatusReport->Get_SubCarrierDemodLock();
    cout << "Symbol synchronization lock status is "
         << pIrafStatusReport->Get_SymbolSyncLock();
    cout << "Production status is "
         << pIrafStatusReport->Get_ProductionStatus();
    break;
  }

  ...
```

4.12.4 GETTING PARAMETERS FROM A PROVIDER SERVICE INSTANCE

After a service instance has been created and configured by the SLE application, and the BIND operation has been performed, the service instance is in ‘bound’ state. In the ‘bound’ state, the service instance is able to process GET-PARAMETER operations. The GET-PARAMETER operation is always sent by the user application. When configuring the GET-PARAMETER operation, the user application must specify the name of the requested parameter by calling the `Set_RequestedParameter()` method. On the provider side, the operation is processed directly by the SLE API and is not forwarded to the application. The provider SLE API, depending on the requested parameter, updates the GET-PARAMETER invocation with the requested parameter value, and sends it back to the user.

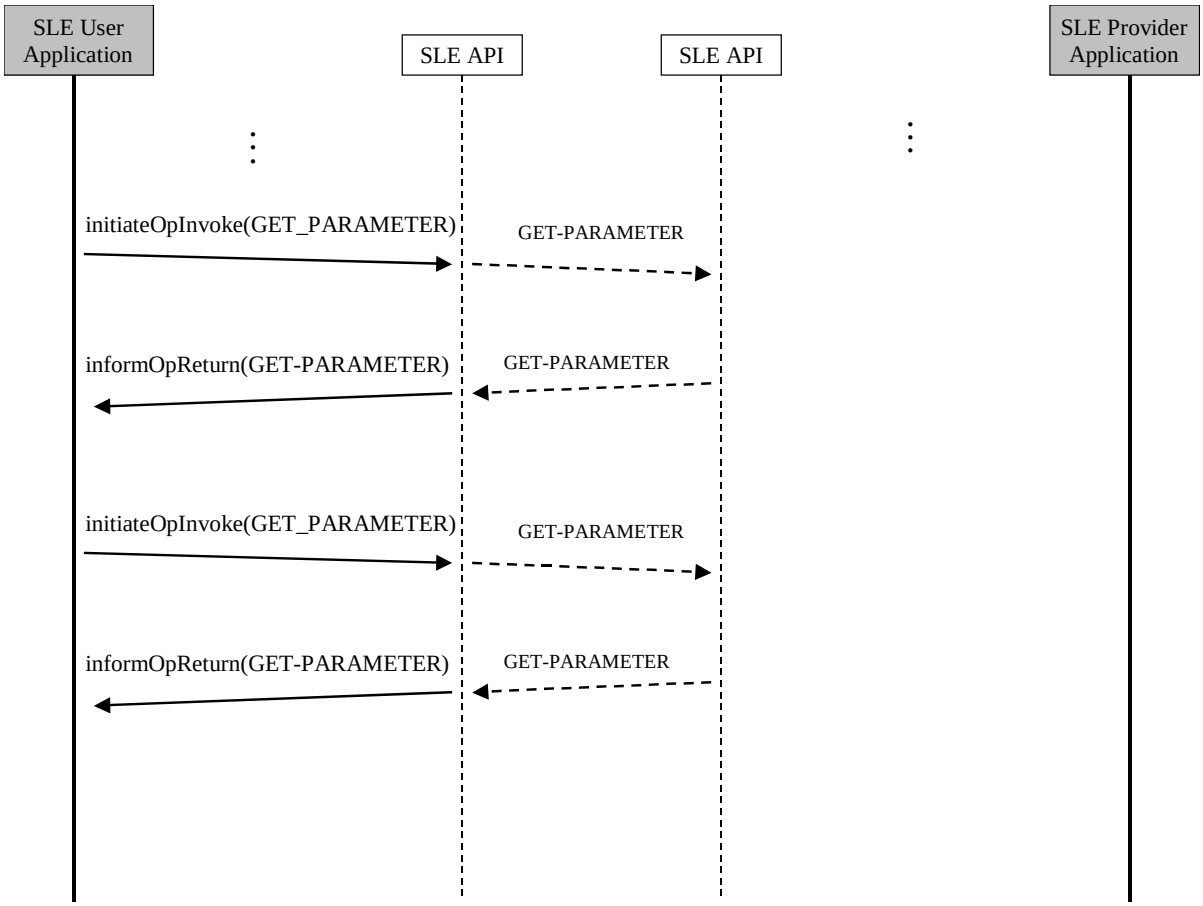

Figure 4-4: Get Parameter Sequence Diagram

NOTES

1 GET-PARAMETER operations are used by the SLE user application to get information on service instance parameter values of the provider service instance. This relies on the fact that the SLE provider application updates its service instance through the service instance update interface, as described in 5.1 and 6.2.

2 Through a GET-PARAMETER operation, the user application can only request the value of one parameter. If several parameters values must be retrieved, several GET-PARAMETER operations must be invoked by the user application.

3 GET-PARAMETER is a service specific SLE operation. It is described in this subsection since its processing is the same for forward and return services. Only the parameters differ between the SLE services.

The following code example shows how a RAF user application should process a GET-PARAMETER return operation. The processing of GET-PARAMETER operation should be the same, regardless of the SLE service type. Only the name of the parameter (and the values) differs from one service to another.

```
switch (pConfirmedOperation->Get_OperationType())
{
  ...

  case sleOT_getParameter:
  {
    // Downcast to the get-parameter operation
    //----------------------------------------
    IRAF_GetParameter *pIrafGetParameter =
                          (IRAF_GetParameter *)pConfirmedOperation;

    // Check the result of the operation
    //----------------------------------
    if (pIrafGetParameter->Get_Result() == sleRES_positive)
    {
      // get the parameter value depending on the parameter name
      //--------------------------------------------------------
      switch (pIrafGetParameter->Get_ReturnedParameter())
      {
        case rafPN_deliveryMode:
          cout << "Provider delivery mode is "
               << pIrafGetParameter->Get_DeliveryMode();
          break;

        case rafPN_latencyLimit:
          cout << "Provider latency limit is "
               << pIrafGetParameter->Get_LatencyLimit();
          break;
        ...
      }
    }
    break;
  }
  ...
```

4.12.5 ABORTING A SERVICE INSTANCE

An SLE PEER-ABORT operation can be invoked by SLE application in any state, except ‘unbound’. Moreover, in some cases, the SLE API also invokes PEER-ABORT operations. In such a case, the abort is sent to the local application, and also to the peer one. The following shows an example where the provider application invokes a PEER-ABORT operation to abort the processing of the service instance. After abort processing, the service instance on the user and provider side is in ‘unbound’ state. The user application is able to restart the processing by invoking a BIND operation.

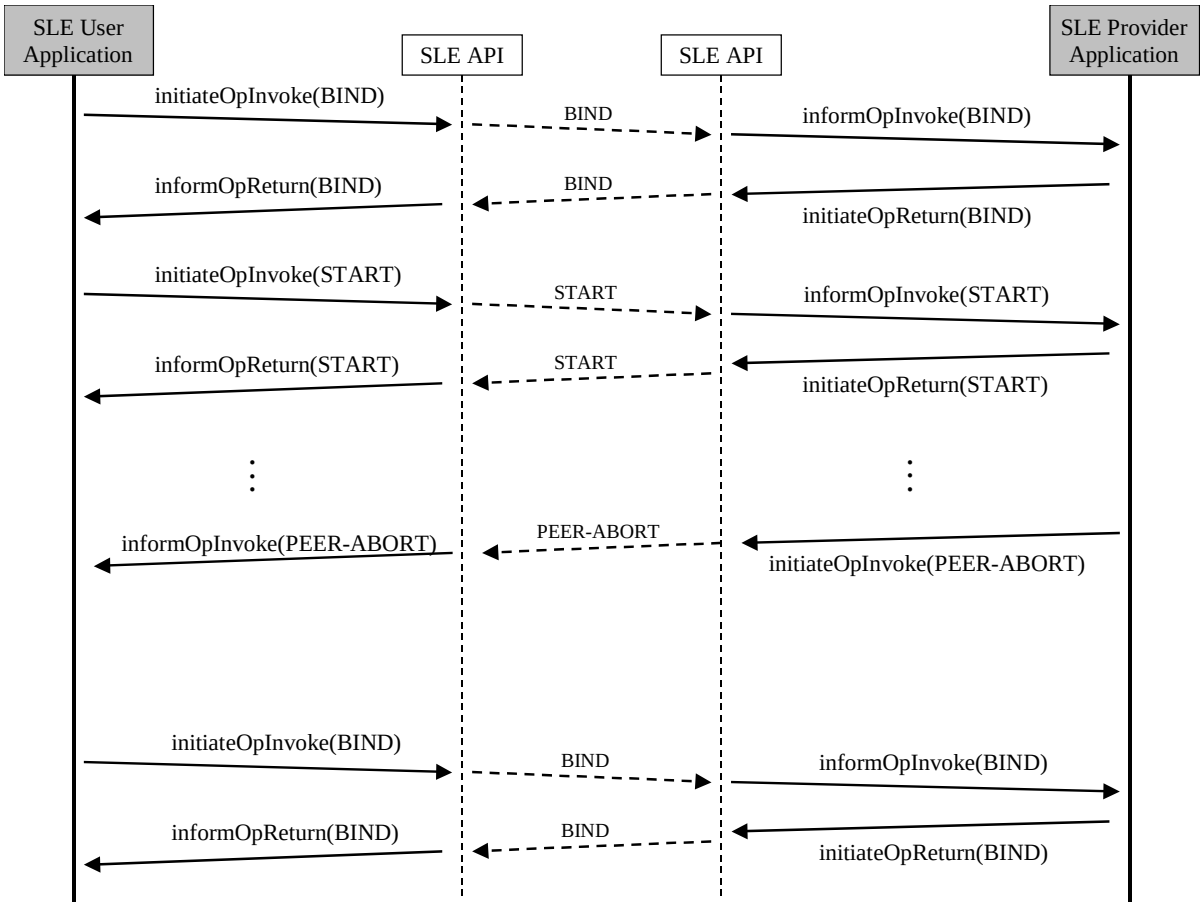

Figure 4-5: Aborting Sequence Diagram

5 SLE RETURN SERVICE APPLICATION

5.1 STATUS INFORMATION AND SERVICE INSTANCE UPDATE

The SLE API internally manages status information for each return service instance. This status information consists of a set of dynamic parameters. An SLE application in the provider role must update the dynamic parameters of the API service instance. For this purpose, the interface `I<SRV>_SIUpdate` must be used.

For return service instance, the dynamic status parameters are:

a) the production status;

b) the lock status; and

c) other service specific information.

The status parameters are updated when the SLE application updates the service instance through the `I<SRV>_SIUpdate` interface, and when SLE operation invocations and returns are processed by the service instance.

In addition, the interface `I<SRV>_SIUpdate` should be used by the application to read and check the configuration parameters. The detailed list of service specific parameters that the provider application must update can be found in reference [12], [13], and [14]. This list is service specific.

5.2 TRANSFER BUFFER

The SLE API fully handles buffering as required by the API Core Specification (reference [10]). For the purpose of sending data from the provider to the user, the SLE API uses a transfer buffer. This transfer buffer stores the TRANSFER-DATA and SYNC-NOTIFY operations received from the provider application, and which must be sent to the user side. This transfer buffer is sent to the user side in the form of a TRANSFER-BUFFER operation. The TRANSFER-BUFFER is a pseudo-operation used to handle the transfer buffer defined in the SLE Recommended Standards (references [4], [5] and [6]).

On the receiving side, the TRANSFER-BUFFER operation is processed by the SLE API. The TRANSFER-DATA and SYNC-NOTIFY operations are extracted from the transfer buffer in the order they were inserted at the provider side, and are delivered to the application.

The provider API service instance decides when to send the transfer buffer, depending on the delivery mode, the transfer buffer size, the release timer, and the operations received from the application.

The provider return application, before using the transfer buffer, must configure:

a) the transfer buffer size via the `Set_TransferBufferSize()` method of the `I<SRV>SIAdmin` interface; and

b) the release timer via the `Set_LatencyLimit()` method of the `I<SRV>SIAdmin` interface.

This configuration must be done on the provider service instance, as described in 4.6.3.

NOTES

1 The configuration of the release timer is only needed in 'online timely' and 'online complete' delivery modes. See explanations in 5.4.1.3.

2 The configuration of the buffer size and the release timer depend on use requirements, and should be made configurable by the SLE application.

5.3 SYNCHRONOUS NOTIFICATION

The provider application must inform the user side of modifications on the provider side by sending notifications. Such notifications are sent in the form of SYNC-NOTIFY operations. For sending theses notifications, the provider application must build SYNC-NOTIFY operations and pass them to the API.

The SYNC-NOTIFY operations can contain the following notifications:

a) data discarded notification: some data have been lost at the provider side;

b) loss of frame synchronization: the frame synchronization has been lost at the provider side;

c) production status changed: the production status changed; and

d) end of data: the provider stopped sending data.

The type of notification is checked by the API service instance together with the configured delivery mode. The list of allowed notification types for a specific delivery mode is detailed in references[12], [13] and [14].

The following example shows how a RAF provider application should create, configure and send a SYNC-NOTIFY operation with 'end of data' notification. The API will append the SYNC-NOTIFY operation to the transfer buffer and send this buffer to the user side. `m_pIrafSIAdmin` is the service instance administrative interface of the RAF service instance on which the operation is invoked. Sequence counting is not used, assuming sequential behavior is used.

```
// Get the service instance operation factory interface
// from the IRAF_SIAdmin interface
//-------------------------------------------------
GUID guidIsleSIOpFactory = IID_ISLE_SIOpFactory_DEF;
ISLE_SIOpFactory * pIsleSIOpFactory = 0;

eResult = m_pIrafSIAdmin->QueryInterface(guidIsleSIOpFactory,
                              (void**)&pIsleSIOpFactory);
if ( FAILED(eResult) )
  //Error handling code

// Create the sync notify operation
//---------------------------------
GUID guidIrafSyncNotify = IID_IRAF_SyncNotify_DEF;
IRAF_SyncNotify * pIrafSyncNotify = 0;

eResult  = pIsleSIOpFactory->CreateOperation (guidIrafSyncNotify,
                                sleOT_syncNotify,
                                (void**)&pIrafSyncNotify);

pIsleSIOpFactory->Release();
pIsleSIOpFactory = 0;

if ( FAILED(eResult) )
  //Error handling code

// Configure the sync notify operation
//------------------------------------
pIrafSyncNotify->Set_EndOfData();

// Get the service instance initiate interface
// from the IRAF_SIAdmin interface
//-----------------------------------------
GUID guidIsleServiceInitiate = IID_ISLE_ServiceInitiate_DEF;
ISLE_ServiceInitiate * pIsleServiceInitiate = 0;

eResult = m_pIrafSIAdmin->QueryInterface(guidIsleServiceInitiate,
                              (void**)&pIsleServiceInitiate);
if ( FAILED(eResult) )
{
  pIrafSyncNotify->Release();
  pIrafSyncNotify = 0;
  //Error handling code
}

// Send the sync notify invoke operation
//--------------------------------------
eResult = pIsleServiceInitiate->InitiateOpInvoke(pIrafSyncNotify, 0);

pIrafSyncNotify->Release();
pIrafSyncNotify = 0;
pIsleServiceInitiate->Release();
pIsleServiceInitiate = 0;

if ( FAILED(eResult) )
  //Error handling code
```

5.4 DATA TRANSFER

5.4.1 PROVIDER SIDE

5.4.1.1 General

At the provider side, the data processing depends on the type of delivery mode, configured by the application. The SLE Recommended Standards define three different delivery modes:

a) Online Timely;

b) Online Complete; and

c) Offline.

The flow control and buffering mechanisms implemented in the SLE API depend on which delivery mode is selected by the provider application.

5.4.1.2 Flow Control for Online Complete and Offline Delivery Modes

In the delivery modes ‘online-complete’ and ‘offline’, the SLE API provides flow control as specified in reference [10]. For the delivery mode ‘online complete’, the API service instance handles the release timer as well; i.e., the service instance starts the release timer when inserting the first PDU into the transfer buffer. The SLE API provider side sends the transfer buffer to the user side when it is full, when the release timer expires (for delivery mode ‘online-complete’), or when a ‘end of data’ notification is appended. The data transfer is suspended when the transmission capacity is exceeded.

The application sends TRANSFER-DATA and SYNC-NOTIFY operation invocations to the SLE API via the `InitiateOpInvoke()` method of the `ISLE_ServiceInitiate` interface:

a) if the received operation can be inserted in the transfer buffer and this one is not yet due to transfer, a positive result code is returned;

b) if the received operation can be inserted in the transfer buffer, and this one is due to transfer:

   1) if no older buffer is already queued, the transfer buffer is queued for transfer and the SLE API returns the positive return code S_OK;

   2) if an older buffer is already queued, the SLE API returns the positive return code `SLE_S_SUSPEND`, indicating that the data transfer shall now be suspended;

c) if the operation is received in a period in which data transfer has been suspended, the SLE API rejects the operation invocation and returns the error code `SLE_E_SUSPENDED`.

When data transmission can be resumed after being suspended, the SLE API informs the application via the method `ResumeDataTransfer()` of the interface `ISLE_ServiceInform`. The number of invocations that can be queued in the transfer buffer is defined by the implementation or can be set in the configuration database of the service element (this is an API implementation specific issue).

5.4.1.3 Buffering in the Delivery Mode Online Timely

For the delivery mode ‘online timely’, the API service instance handles the release timer and discarding of buffers as defined by the Recommended Standards for return SLE 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 an ‘end of data’ SYNC-NOTIFY operation is appended to the buffer, the service instance forwards the transfer buffer for sending to the user side.

NOTES

1 The API internally may discard the previously stored transfer buffer if this one has not yet been sent to the user side. In such a case, the API service instance inserts a notification ‘data discarded due to excessive backlog’ at the beginning of the transfer buffer.

2 The SLE API maintains a counter of frames transmitted for use in the status report. This count does not include frames discarded and can be used by the application to determine how many frames were discarded.

5.4.2 USER SIDE

On the user side, the return link SLE application receives the SLE operations as described in 4.7.7. The TRANSFER-DATA and SYNC-NOTIFY operations, extracted from the TRANSFER-BUFFER operation by the API, are passed to the application in the sequence they have been stored at the provider side.

The following example shows how a RAF user application should process a SYNC-NOTIFY invoke operation.

```
switch (pOperation->Get_OperationType())
{
  ...
  case sleOT_syncNotify:
  {
    // Downcast to the sync-notify operation
    //--------------------------------------
    IRAF_SyncNotify *pIrafSyncNotify = (IRAF_ SyncNotify *)pOperation;
```

```
  // get the parameter value depending on the parameter name
  //-------------------------------------------------------
  switch (pIrafSyncNotify->Get_NotificationType())
  {
    case rafNT_lossFrameSync:
      cout << "Provider lost frame synchronisation at "
           << pIrafSyncNotify->Get_LossOfLockTime();
      break;

    case rafNT_productionStatusChange:
      cout << "Provider production status changed to "
           << pIrafSyncNotify->Get_ProductionStatus();
      break;

    case rafNT_excessiveDataBacklog:
      cout << "Provider lost data ";
      break;

    case rafNT_endOfData:
      cout << "End of data";
      break;

    default:
      cout << "Invalid SYNC-NOTIFY operation received";
      break;
  }
  break;
}
```

5.5 ONLINE DATA TRANSFER EXAMPLE WITHOUT 'END OF DATA'

The following figure provides a scenario describing a SLE RAF provider application sending RAF TRANSFER-DATA invocations, in 'online timely' or 'online complete' delivery mode. The first TRANSFER-BUFFER is sent to the user side by the SLE API because it is full, the second one because the release timer expires, the third one because the user requested a stop of the processing.

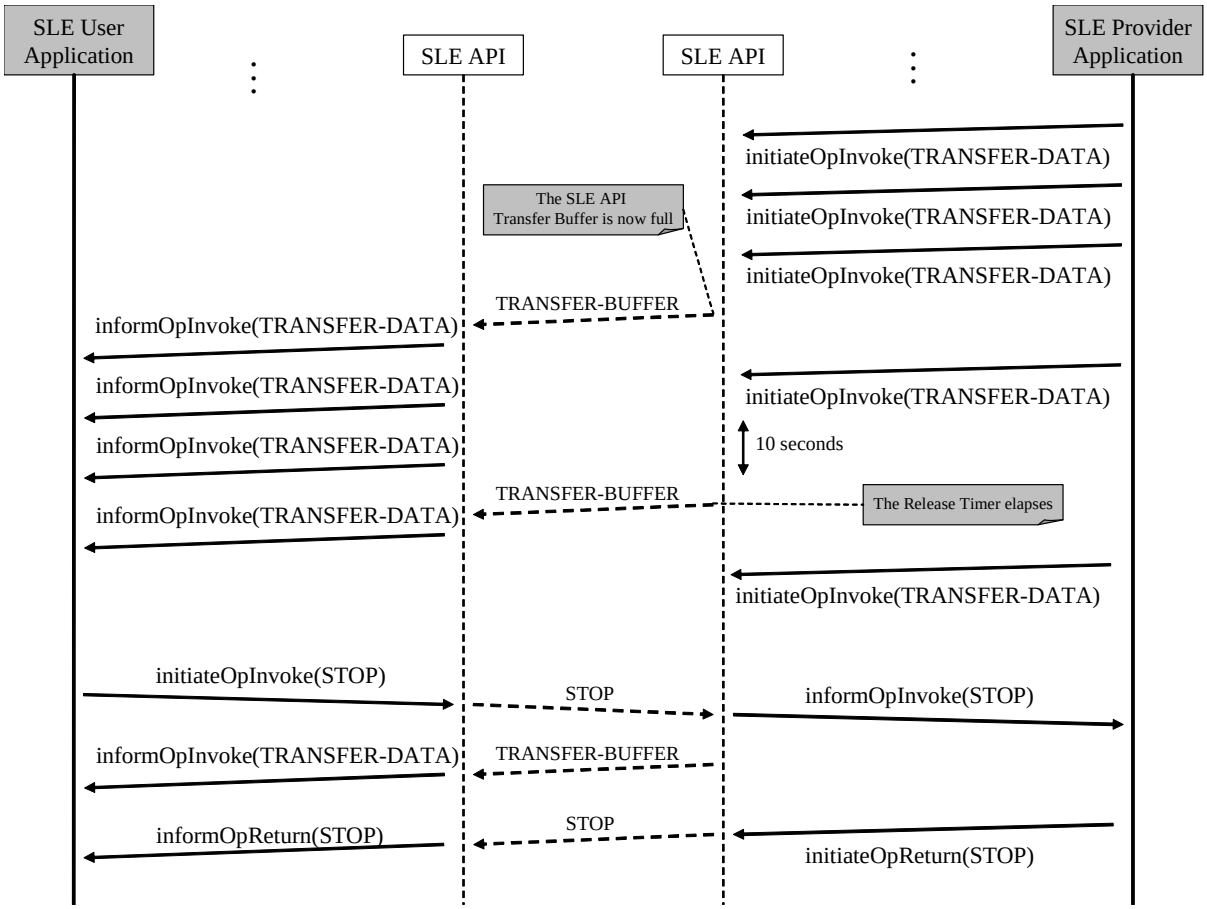

Figure 5-1: Online Data Transfer Sequence Diagram without 'End of Data'

NOTE – The release timer is used in both delivery modes 'online timely' and 'online complete'.

5.6 ONLINE DATA TRANSFER EXAMPLE WITH 'END OF DATA'

The following figure provides a scenario describing a SLE RAF provider application sending RAF TRANSFER-DATA invocations, in 'online timely' or 'online complete' delivery mode. The first TRANSFER-BUFFER is sent to the user side by the SLE API because it is full, the second one because the release timer expires, the third one because the application sent a 'end of data' notification.

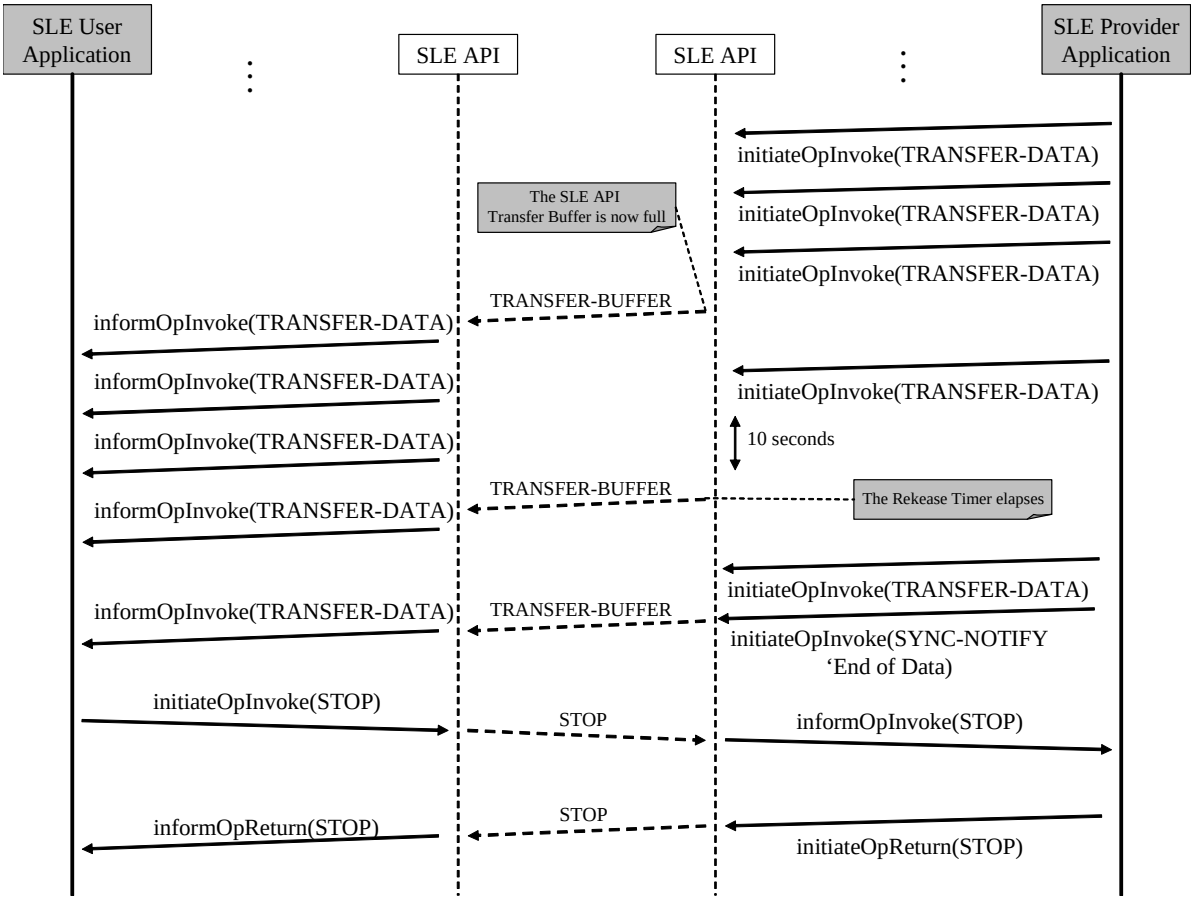

Figure 5-2: Online Data Transfer Sequence Diagram with 'End of Data'

NOTE – The SYNC-NOTIFY operation 'End of Data' is used in all delivery modes.

5.7 OFFLINE DATA TRANSFER EXAMPLE

The following figure provides a scenario describing a SLE RAF provider application sending RAF TRANSFER-DATA invocations, in 'offline' delivery mode. The first and second TRANSFER-BUFFER are sent to the user side by the API because they are full, the third one because the application sent a 'end of data' notification.

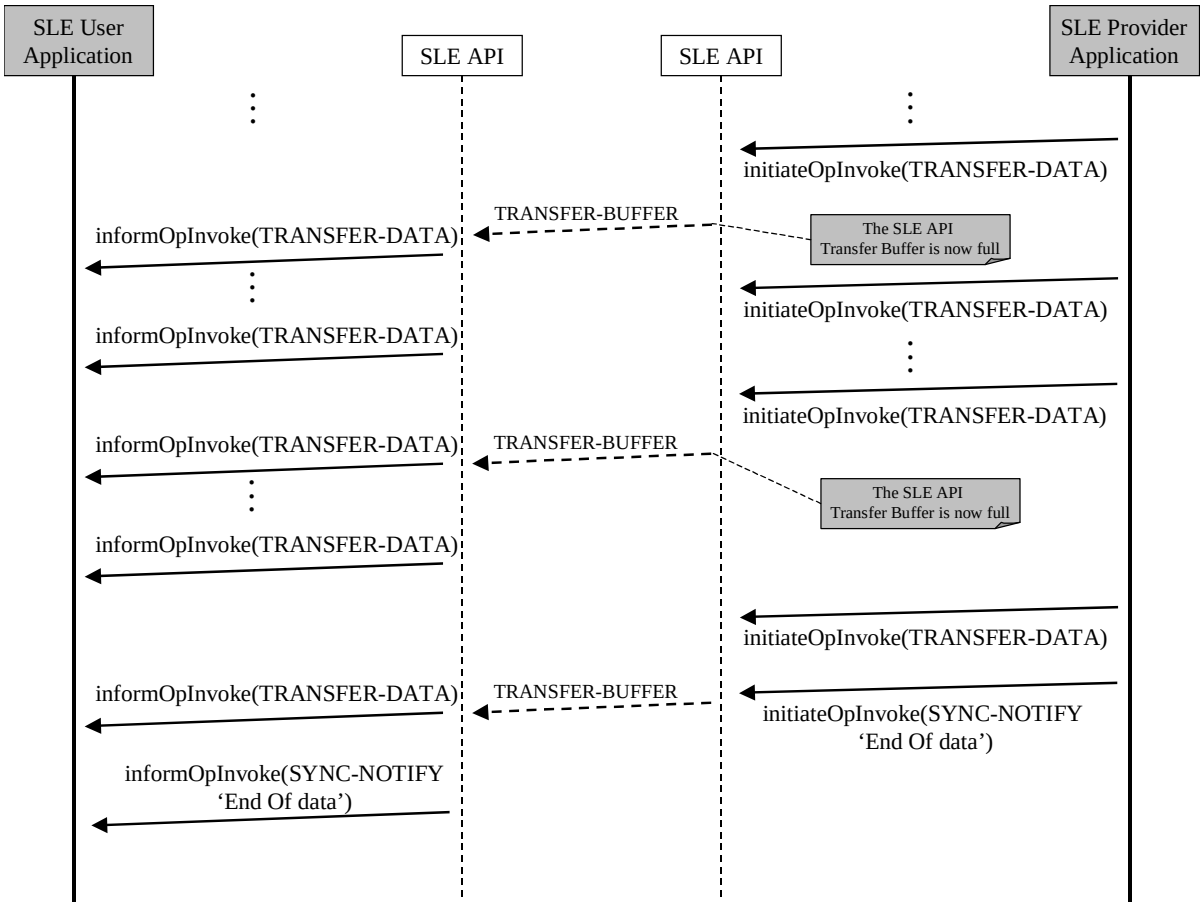

Figure 5-3: Offline Data Transfer Sequence Diagram

6 SLE FORWARD SERVICE APPLICATION

6.1 STATUS INFORMATION

The SLE API internally manages status information for each forward service instance. This status information consists of a set of dynamic parameters. For forward service instance, the dynamic status parameters are:

a) the production status;

b) the amount of buffer space available;

c) telecommand processing information;

d) telecommand radiation information; and

e) other service specific information.

The status parameters are updated when the SLE application updates the service instance through the `I<SRV>_SIUpdate` interface (see 6.2), and when SLE operation invocations and returns are processed by the service instance.

NOTE — The list of status parameters, and the description on how these parameters are updated is described in references [15] and [16].

6.2 FORWARD SERVICE INSTANCE UPDATE

6.2.1 INTRODUCTION

SLE application in the provider role must inform the SLE API service instance of specific events in the production telecommand process. For this purpose, the interface `I<SRV>_SIUpdate` must be used. These events should be reported to the service instance during its complete lifetime. Depending on the type of event reported by the SLE application, a notification may be sent by the SLE API to the user side. Such notifications are sent in the form of ASYNC-NOTIFY operations.

NOTE — For sending notifications to the user side, the provider application must not build ASYNC-NOTIFY operations and pass them to the SLE API. Instead, it must update the SLE API service instance by calling the appropriate methods of the `I<SRV>_SIUpdate` interface, setting the ‘notify’ Boolean of these methods to ‘true’.

The interface `I<SRV>_SIUpdate` should also be used by the application to read and check the configuration parameters. The detailed list of service specific parameters that the provider application must update can be found in references [15] and [16]. This list is service specific.

The following tables describe how an SLE application should update CLTU and FSP service instance when production events occur. In each table is presented:

a) the production event that shall be reported by the application to the SLE API;

b) the name of the method (of the `I<SRV>_SIUpdate` interface) the application shall call to report the event;

c) the arguments to provide;

d) the list of status parameters updated by the SLE API when the event is reported; and

e) the notification sent by the SLE API to the user side (if any). The notification type depends on the method arguments and partially on the value of the production status. The SLE API behavior also depends on configuration parameter ‘notification mode’.

Table 6-1: CLTU Service—Production Events Reported via the Interface `ICLTU_SIUpdate`

Event	Method	Arguments	Status parameters updated	Notification sent
Radiation of a CLTU started	`CltuStarted`	CLTU identification radiation start time available buffer size	CLTU identification last processed radiation start time CLTU status number of CLTUs processed available buffer size	none
Radiation of a CLTU completed	`CltuRadiated`	radiation start time radiation stop time[2]	CLTU identification last OK radiation stop time CLTU status number of CLTUs radiated	CLTU radiated
Radiation of a CLTU could not be started because the latest radiation time expired or the production status was interrupted	`CltuNotStarted`	CLTU identification failure reason available buffer size	CLTU identification last processed radiation start time CLTU status number of CLTUs processed available buffer size	SLDU expired production interrupted
The CLTU buffer is empty.	`BufferEmpty`		available buffer size	buffer empty
The production status changed (with or without affecting a CLTU being radiated)	`ProductionStatusChange`	production status[3] available buffer size	production status available buffer size	production interrupted production halted production operational
The uplink status changed	`Set_UplinkStatus`	uplink status	uplink status	none
Processing of a thrown event completed	`EventProcCompleted`	event id event proc result		action list completed action list not completed event condition evaluated to false

[2] The start time is an optional parameter that can be supplied if the exact start time is known only after radiation of the CLTU. In such a case the start time passed to the method `CktuStarted` should be the best available estimate.

[3] When the production status is set to 'interrupted', the SLE API sends immediately a notification to the user side if the configurable parameter 'notification mode' is set to 'immediate'. Otherwise (configurable parameter 'notification mode' set to 'deferred') the SLE API waits until a CLTU is ready to be radiated before sending the notification.

Table 6-2: FSP Service—Production Events Reported via the Interface `IFSP_SIUpdate`

Event	Method	Arguments	Status parameters updated	Notification sent
Processing of a packet started	`PacketStarted`	packet-id transmission-mode start time available buffer size	packet id last processed production start time packet status number of AD packets processed[1] number of BD packets processed[2] packet buffer available	packet processing started
Radiation of a packet completed	`PacketRadiated`	packet-id transmission mode radiation time	packet id last OK[2] packet status[2] production stop time[2] number of AD packets radiated[1] number of BD packets radiated[2]	packet radiated
All segments of an AD packet acknowledged via the CLCW	`PacketAcknowledged`	packet-id acknowledge time	packet id last OK packet status production stop time number of packets acknowledged	packet acknowledged
The packet buffer is empty	`BufferEmpty`		packet buffer available	buffer empty
Processing of a packet could not be started because – the latest production time expired; – the production status was interrupted; or – the required transmission mode was not available	`PacketNotStarted`	packet id transmission mode start time failure reason affected packets list available buffer size	packet id last processed packet status production start time packet buffer available	SLDU expired production interrupted transmission mode mismatch
The production status changed	`ProductionStatusChange`	production status affected packets list[4] fop alert[5] available buffer size	production status packet status[3] packet buffer available	production operational production interrupted production halted transmission mode capability change transmission mode mismatch

Event	Method	Arguments	Status parameters updated	Notification sent
The VC was aborted by a directive	`VCAborted`	affected packets list[4] available buffer size	packet status[3] production status packet buffer available	VC aborted
The service instance with directive invocation capability is no longer connected	`NoDirectiveCapability`			no invoke directive capability on this VC
A service instance with directive invocation capability has bound	`DirectiveCapability Online`		directive invocation online	invoke directive capability on this VC established
Processing of a directive completed	`DirectiveCompleted`	directive id result fop alert[6]		positive confirm response to directive negative confirm response to directive
Processing of a thrown event completed	`EventProcCompleted`	event id event proc result		action list completed action list not completed event condition evaluated to false

1. If the transmission mode is sequence controlled.
2. If the transmission mode is expedited.
3. If the packet id last processed is contained in the affected packets list argument.
4. If no packets were affected, the list is empty.
5. Only needed in case of a transmission mode capability change.
6. Only needed in case of a negative result.

6.2.2 PRODUCTION STATUS UPDATE

The SLE provider forward application must manage an internal production status, which reflects the state of the telecommand production engine. The initial state of the production status must be provided to the SLE API via the `Set_InitialProductionStatus()` method of the `I<SRV>_SIAdmin` interface. Moreover, the application must inform the API about every production status state change.

The way the application informs the API on production status changes is service specific. For instance, for the CLTU service, it depends on whether the change of production status occurs before or during the radiation of a CLTU. If the change occurs before the radiation of the next CLTU has started, the `ProductionStatusChange()` method of the `ICLTU_SIUpdate` interface must be used, followed by a call to `CltuNotStarted()` when the application attempts to radiate the CLTU. If the change affects a CLTU being radiated, only the `ProductionStatusChange()` method must be used – if the method `CltuStarted()` has been called before, the SLE API knows that a CLTU is affected.

The following example shows how a CLTU provider service instance updates the production status, when this one changes to 'interrupted' state, without affecting CLTU radiation. The last parameter of the `ProductionStatusChange()` method is set to 'true' with the effect that a notification is sent to the user side if the configured notification type parameter is 'immediate'. `m_pCltuSIUpdate` is a reference to the service instance update interface of the CLTU service instance. `m_bufferAvailable` is an object attribute containing the current amount of buffer available.

```
// Update the service instance with the new production status
//-------------------------------------------------------------
bool sendNotification = true;

eResult = m_pCltuSIUpdate->ProductionStatusChange(cltuPS_interrupted,
                               m_bufferAvailable, sendNotification);

if ( FAILED(eResult) )
  //Error handling code
```

6.2.3 BUFFER AVAILABLE

The SLE provider forward application must manage buffers in order to temporarily store the telecommand to be radiated. At startup, the initial buffer size must be provided to the SLE API, via the `Set_MaximumBufferSize()` method of the `I<SRV>_SIAdmin` interface. Moreover, the application, during telecommand processing and radiation, must inform the API about the current amount of buffers available. The buffer size is reset to its maximum value by the SLE API when a STOP invocation is received from the user side, or when an abort occurs.

When all buffers are empty, the SLE provider application must inform the API, using the `BufferEmpty()` method of the `I<SRV>_SIUpdate` interface.

The following example shows how a CLTU provider service instance updates the amount of buffer available, informs the API on this amount, and sends a notification as required. The example shows the code executed when the radiation of a CLTU starts. The `removeBuffer()` method removes a CLTU from the buffer, and returns the size of the removed CLTU. `m_pCltuSIUpdate` is a reference to the service instance update interface of the CLTU service instance. `m_bufferAvailable` is a class attribute containing the current amount of buffer available. `m_maxBufferSize` is a class attribute containing the maximum amount of buffer.

```
// radiation of CLTU cltuId has started
// remove this CLTU from the buffer
// ---------------------------------
CLTU_BufferSize size = removeCLTU(cltuId);

// increase the amount of available buffer
//----------------------------------------
m_bufferAvailable += size;

// inform the API on the start of radiation (providing radiation
// start time) and on the new amount of buffer available
//-------------------------------------------------------------
eResult = m_pCltuSIUpdate->CltuStarted(cltuId, radiationStartTime,
                                m_bufferAvailable);
if ( FAILED(eResult) )
  //Error handling code

// Check if the buffer is empty
//-----------------------------
if (m_bufferAvailable == m_maxBufferSize)
{
  // send a buffer empty notification
  // --------------------------------
  eResult = m_pCltuSIUpdate->BufferEmpty(true);

  if ( FAILED(eResult) )
  //Error handling code
}
```

6.2.4 EXAMPLE OF ASYNC-NOTIFY OPERATIONS

The following example shows how ASYNC-NOTIFY operations are sent when a CLTU provider application updates the service instance. Several notifications are sent by the API service instance:

a) the ‘CLTU radiated’ notification is sent because the provider application, after having radiated a CLTU, has called the `CltuRadiated()` method with the notify parameter set to ‘true’;

b) the ‘buffer empty’ notification is sent because the provider application called the `BufferEmpty()` method with the notify parameter set to ‘true’; and

c) the ‘production halted’ notification is sent because the provider application called the `ProductionStatusChange()` method to set the production status to halted, with the notify parameter set to ‘true’.

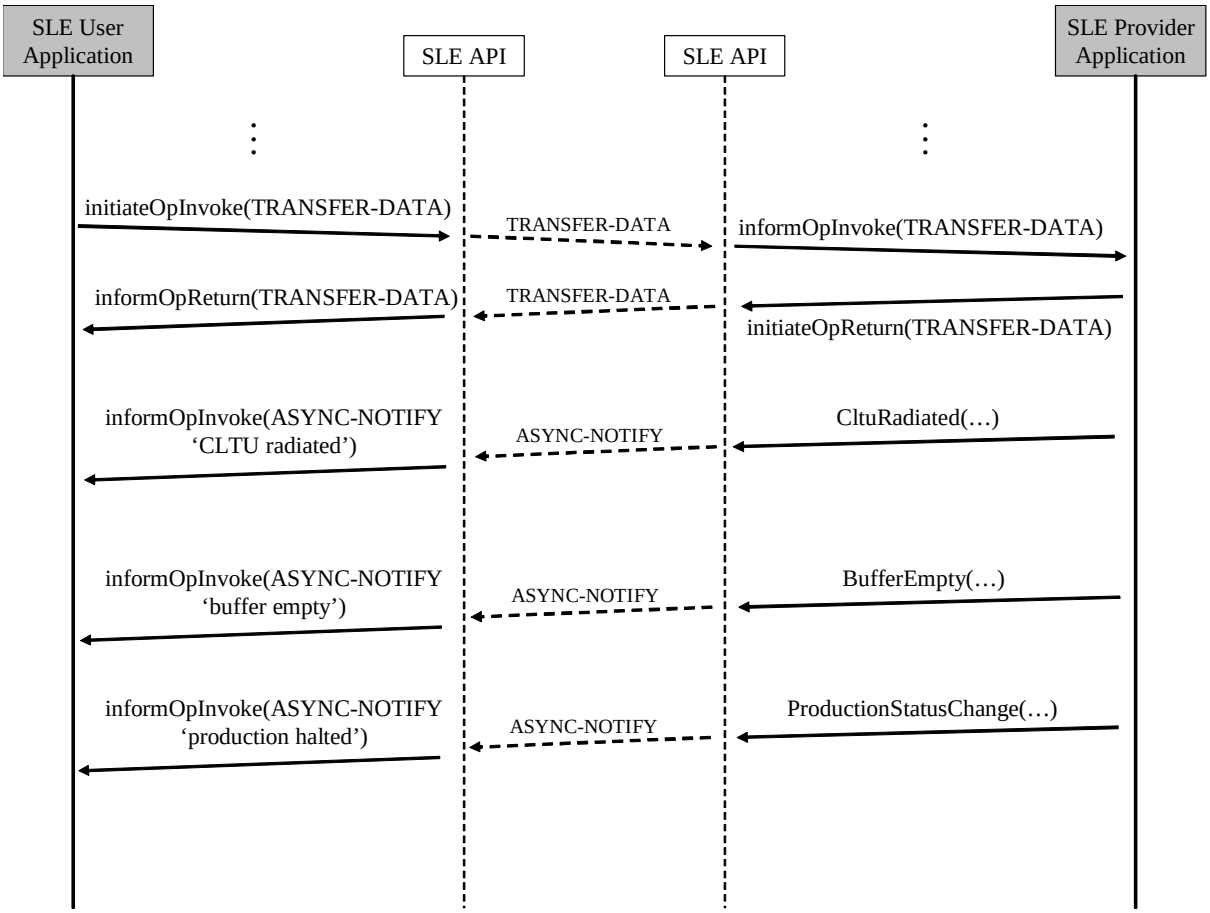

Figure 6-1: Asynchronous Notification Sequence Diagram

6.3 DATA TRANSFER

6.3.1 GENERAL

In order to send telecommands to the provider side, the forward user application must build TRANSFER-DATA operations from the telecommands, and pass them to the API service instance, as described in 4.7.6. For this purpose, the method `InitiateOpInvoke()` is used.

6.3.2 FLOW CONTROL

The user forward service instance provides flow control for TRANSFER-DATA invocations. When receiving a TRANSFER-DATA invocation from the application, the service element forwards it to the proxy. If the proxy cannot transfer the invocation immediately, the `InitiateOpInvoke()` method returns the code `SLE_S_SUSPEND` to the application requesting it to suspend data transfer. When data transmission can be resumed, the API service instance informs the application via the method `ResumeDataTransfer()` of the interface `ISLE_ServiceInform`. A TRANSFER-DATA invocation from the application is rejected by the SLE API with return code `SLE_E_SUSPENDED` when it is sent after suspend and before data transfer has resumed.

The following figure provides a scenario describing a SLE CLTU user application invoking CLTU-TRANSFER-DATA operations. When the SLE API buffer is full and the user application invokes a CLTU-TRANSFER-DATA operation, the SLE API informs the user application of a suspension of the data transfer (the method `InitiateOpInvoke()` returns the result code `SLE_S_SUSPEND`). When the user API has sent some CLTU-TRANSFER-DATA operations to the provider, the data transfer can resume. The SLE API calls the method `ResumeDataTransfer()` of the application, which resumes sending of CLTU-TRANSFER-DATA operations.

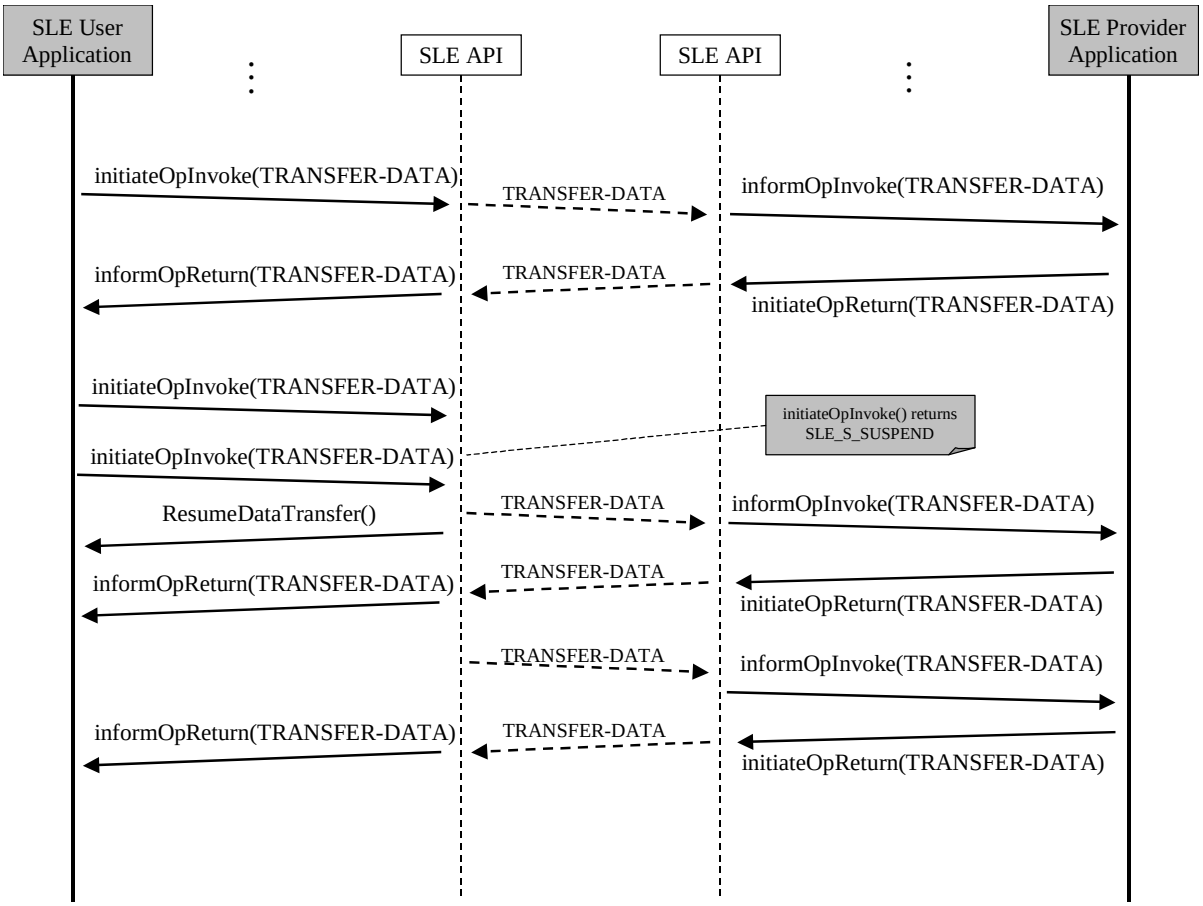

Figure 6-2: Data Transfer Sequence Diagram

The following example shows how a CLTU user application sends a CLTU, taking care of the code returned by the `InitiateOpInvoke()` method of the `ISLE_ServiceInitiate` interface. The function `waitResumeDataTransfer()` waits until a resume of data transfer is sent by the API service instance. `m_pIsleServiceInitiate` is a reference to the service instance initiate interface of the CLTU service instance. The sequence counter is not used since sequential behavior is assumed. `pOp` is a reference to the `ISLE_Operation` interface of the CLTU operation.

```
// send the CLTU operation to the service instance
//-----------------------------------------------
eResult = m_pIsleServiceInitiate->InitiateOpInvoke(pOp, 0);

if ( FAILED(eResult) )
  //Error handling code
```

```
// Check the result code for flow control purpose
//-----------------------------------------------
if (eResult == SLE_S_SUSPEND)
{
  // wait until the service instance informs about resume
  // of data transfer
  //-------------------------------------------------
  bool waitResult = waitResumeDataTransfer();

  if (waitResult == false )
    //processing error - handle error
}
```

NOTE – When the SLE application notices that data transfer must be suspended (`InitiateOpInvoke()` returns `SLE_S_SUSPEND`), the application should stop sending further TRANSFER-DATA operations (theses operations would be rejected with a return code set to `SLE_E_SUSPENDED`). The way the SLE application shall wait for resuming the data transfer depends on the SLE application implementation.

6.3.3 BLOCKED STATE OF THE SERVICE INSTANCE

When a telecommand cannot be radiated because the production status becomes non-operational or because the latest radiation start time expired, the service instance on the user side becomes blocked and further TRANSFER-DATA invocation are rejected by the SLE API with the diagnostic ‘unable to process’.

In order to recover from this block state, the SLE user application must invoke a STOP operation followed by a START operation.

6.3.4 TRANSFER-DATA RETURN

When the provider side detects a problem while performing a TRANSFER-DATA operation invoke, it has to return a TRANSFER-DATA operation with negative result. Reference [7] and [8] requires that the provider shall insert the next expected telecommand identification and the available buffer size for the TRANSFER-DATA operation return. Because the API service element cannot know what values to insert, the TRANSFER-DATA operation return with negative result must be configured and invoked by the SLE provider application.

When the provider application receives a TRANSFER-DATA invocation from the SLE API, it must:

a) Check the result of the TRANSFER-DATA invocation. A negative result indicates that the SLE API has detected a problem while performing the operation.

b) If the result is negative, set the next expected telecommand identification and the available buffer size in the TRANSFER-DATA operation, and return the operation to the SLE API (the operation must not be processed).

c) If the result is positive, process it.

NOTE – The same mechanism also applies for THROW-EVENT operation.

6.4 PROTOCOL ABORT

When communication problems are detected, the SLE API aborts the association between the user and the provider and informs the local application using the method `ProtocolAbort()`, as described in 4.8. When receiving a protocol abort, a provider forward application must first examine the value of the configurable parameter 'protocol abort mode':

a) if 'protocol abort mode' is set to 'continue', the provider application must continue the production of telecommands, and buffered telecommands must not be discarded; and

b) if 'protocol abort mode' is set to 'flush', the provider application must immediately stop production of telecommands. In case of one telecommand being in the process of being radiated, that one telecommand shall continue to be processed. All buffered telecommands must be discarded.

ANNEX A

GLOSSARY

This annex provides a glossary of important terms used throughout the report. The explanations provided in this annex are informative in nature. Normative definitions of these terms can be found in the SLE Reference Model (reference [3]), the Recommended Standards for SLE transfer services (references [4], [5], [6], [7], and [8]), and the Recommended Practice documents for the SLE API (references [10], [12], [13], [14], [15] and [16]).

A1 APPLICATION PROGRAM INTERFACE (API)

An application program interface (API) is the specific method prescribed by a computer operating system or by another application program by which a programmer writing an application program can make requests of the operating system or another application.

In the context of the SLE API, the term is also used to refer to the set of services that can be requested via the interface and to the software that implements these services.

A2 API PROXY

The API Proxy is a component of the SLE API, which encapsulates all technology specific data communication interfaces. The API core specification defines the interfaces and the functionality of the proxy in a technology independent manner. Proxy components supporting specific technologies are defined in the Recommended Standard on technology mapping.

A3 API SERVICE ELEMENT

The API Service Element is a component of the SLE API, which implements lower level aspects of the SLE transfer service protocol as far as these are technology independent. The API Service Element uses the services and interfaces provided by the API Proxy.

A4 ASSOCIATION

An association is a cooperative relationship between an SLE service-using application entity and an SLE service-providing application entity. An association is formed by the exchange of SLE protocol control information through the use of an underlying communications service.

A5 BINDING

The act of establishing an association between a SLE service user and a SLE service provider is called binding. Following the general SLE approach, binding is specified in the form of a BIND operation, which is invoked by one entity and accepted (or rejected) by the other entity. If the underlying communication service is connection oriented, binding might include establishment of a data communications connection.

A6 INTERFACE

An interface is a collection of semantically related functions (in the sense of a programming language) providing access to the services of an object or a component. Interfaces only contain an abstract specification of functions and do not contain any data declarations or implementation details.

A7 OPERATION

An operation is a procedure or task that one entity (the invoker) can request of another (the performer) through an association. The terms invoker and performer are used to describe the interaction between two entities as the operations that constitute the service occur. Some operations are invoked by the service user and performed by the service provider, whereas others are invoked by the service provider and performed by the service user.

An example for a SLE transfer service operation is TRANSFER-DATA by which one annotated space-link data unit is passed from the invoker to the performer. For return services, TRANSFER-DATA is invoked by the service provider and performed by the service user. For forward services, TRANSFER-DATA is invoked by the service user and performed by the service provider.

Operations can be confirmed or unconfirmed. A confirmed operation is an operation that requires the performer to return a report of its outcome to the invoker, while the result of an unconfirmed operation is not reported to the invoker.

A8 SOFTWARE COMPONENT

A software component is a software module providing a well-defined service via one or more interfaces. In this Report, the term is only used to refer to the API components identified in 0. API components must conform to a set of design and implementation conventions defined in the API core specification (reference [10]) and referred to as SLE Simple Component Model.

The following, more general definition of the term is taken from reference [19]:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

A9 SERVICE INSTANCE

Space Link Extension services require that transfer service provision be fully specified and scheduled by management in advance. A specific service scheduled by a service provider for a specified service user is called a service instance. Service instances are made available by a SLE service provider during the scheduled provision period. A SLE service user can actually use the service once or several times during this period. For every instance of use, the SLE service user establishes an association with the SLE service provider, by means of the BIND operation.

A10 SERVICE USER AND SERVICE PROVIDER

An entity that offers a service to another is called a service provider (provider). The other entity is called a service user (user). The terms user and provider are used to distinguish the roles of two interacting entities. In the SLE context, when two entities are involved in provision of a service, the entity closer to the space link is considered to be the provider of the service, and the object further from the space link is considered to be the user.

ANNEX B

ACRONYMS

This annex expands the acronyms used throughout this Report.

API	Application Program Interface
CCSDS	Consultative Committee for Space Data Systems
CLTU	Command Link Transmission Unit
COM	Component Object Model
CPU	Central Processing Unit
FOP	Frame Operation Procedure
FSP	Forward Space Packet
GUID	Globally Unique Identifier
IEC	International Electrotechnical Commission
IID	Interface Identifier
IP	Internet Protocol
ISP1	Internet SLE Protocol One
ISO	International Organization for Standardization
MC	Master Channel
OMG	Object Management Group
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
RAF	Return All Frames
RCF	Return Channel Frames
RFC	Request For Comments
ROCF	Return Operational Control Field
SCM	Simple Component Model

SLDU	Space Link Data Unit
SLE	Space Link Extension
UML	Unified Modeling Language
UTC	Coordinated Universal Time
VC	Virtual Channel