CCSDS
The Consultative Committee for Space Data Systems

CCSDS Historical Document

This document's Historical status indicates that it is no longer current. It has either been replaced by a newer issue or withdrawn because it was deemed obsolete. Current CCSDS publications are maintained at the following location:

http://public.ccsds.org/publications/

CCSDS HISTORICAL DOCUMENT

Recommendation for Space Data System Practices

SPACE LINK EXTENSION— APPLICATION PROGRAM INTERFACE FOR THE FORWARD SPACE PACKET SERVICE

RECOMMENDED PRACTICE

CCSDS 916.3-M-1

MAGENTA BOOK
October 2008

CCSDS HISTORICAL DOCUMENT

CCSDS HISTORICAL DOCUMENT

Recommendation for Space Data System Practices

SPACE LINK EXTENSION— APPLICATION PROGRAM INTERFACE FOR THE FORWARD SPACE PACKET SERVICE

RECOMMENDED PRACTICE

CCSDS 916.3-M-1

MAGENTA BOOK
October 2008

CCSDS HISTORICAL DOCUMENT

CCSDS HISTORICAL DOCUMENT

AUTHORITY

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

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

This document is published and maintained by:

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

CCSDS HISTORICAL DOCUMENT

STATEMENT OF INTENT

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

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

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

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

CCSDS HISTORICAL DOCUMENT

FOREWORD

This document is a technical **Recommended Practice** 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 **Recommended Practice** specifies service type-specific extensions of the Space Link Extension Application Program Interface for Transfer Services specified by CCSDS (reference [4]). It allows implementing organizations within each Agency to proceed with the development of compatible, derived Standards for the ground systems that are within their cognizance. Derived Agency Standards may implement only a subset of the optional features allowed by the **Recommended Practice** and may incorporate features not addressed by the **Recommended Practice**.

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

http://www.ccsds.org/

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

CCSDS HISTORICAL DOCUMENT

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.

CCSDS HISTORICAL DOCUMENT

DOCUMENT CONTROL

Document	Title	Date	Status
CCSDS 916.3-M-1	Space Link Extension—Application Program Interface for the Forward Space Packet Service, Recommended Practice, Issue 1	October 2008	Original issue

CCSDS HISTORICAL DOCUMENT

CONTENTS

Section Page

1 INTRODUCTION ... 1-1

1.1 PURPOSE ... 1-1
1.2 SCOPE ... 1-1
1.3 APPLICABILITY ... 1-1
1.4 RATIONALE ... 1-1
1.5 DOCUMENT STRUCTURE ... 1-2
1.6 DEFINITIONS, NOMENCLATURE, AND CONVENTIONS ... 1-4
1.7 REFERENCES ... 1-7

2 OVERVIEW ... 2-1

2.1 INTRODUCTION ... 2-1
2.2 PACKAGE FSP SERVICE INSTANCES ... 2-1
2.3 PACKAGE FSP OPERATIONS ... 2-16
2.4 SECURITY ASPECTS OF THE SLE FORWARD SPACE PACKET (FSP) TRANSFER SERVICE ... 2-18

3 FSP SPECIFIC SPECIFICATIONS FOR API COMPONENTS ... 3-1

3.1 API SERVICE ELEMENT ... 3-1
3.2 SLE OPERATIONS ... 3-22
3.3 SLE APPLICATION ... 3-23
3.4 SEQUENCE OF DIAGNOSTIC CODES ... 3-25

ANNEX A FSP SPECIFIC INTERFACES (Normative) ... A-1
ANNEX B ACRONYMS (Informative) ... B-1
ANNEX C INFORMATIVE REFERENCES (Informative) ... C-1

Figure

1-1 SLE Services and SLE API Documentation ... 1-3
2-1 FSP Service Instances ... 2-2
2-2 FSP Operation Objects ... 2-17

Table

2-1 Production Events Reported via the Interface IFSP_SIUpdate ... 2-5
2-2 FSP Configuration Parameters Handled by the Service Element ... 2-9
2-3 FSP FOP Parameters Handled by the Service Element ... 2-10

CCSDS HISTORICAL DOCUMENT

CONTENTS (continued)

Table Page

2-4 FSP Status Parameters Handled by the Service Element 2-11
2-5 FSP Production Status 2-13
2-6 Mapping of FSP Operations to Operation Object Interfaces...................................... 2-18

CCSDS HISTORICAL DOCUMENT

1 INTRODUCTION

1.1 PURPOSE

The Recommended Practice *Space Link Extension—Application Program Interface for Transfer Services—Core Specification* (reference [5]) specifies a C++ API for CCSDS Space Link Extension Transfer Services. The API is intended for use by application programs implementing SLE transfer services.

Reference [5] defines the architecture of the API and the functionality that is independent of specific SLE service types.

The purpose of this document is to specify extensions to the API needed for support of the Forward Space Packet Service defined in reference [3].

1.2 SCOPE

This specification defines extensions to the SLE API in terms of:

a) the FSP-specific functionality provided by API components;

b) the FSP-specific interfaces provided by API components; and

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

It does not specify:

a) individual implementations or products;

b) the internal design of the components; and

c) the technology used for communications.

This Recommended Practice defines only interfaces and behavior that must be provided by implementations supporting the Forward Space Packet service in addition to the specification in reference [5].

1.3 APPLICABILITY

The FSP Application Program Interface specified in this document supports version 1 of the FSP service, as specified in reference [3].

1.4 RATIONALE

This Recommended Practice specifies the mapping of the Forward Space Packet service specification to specific functions and parameters of the SLE API. It also specifies the

CCSDS HISTORICAL DOCUMENT

distribution of responsibility for specific functions between SLE API software and application software.

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

a) application software using the SLE API and SLE API software implementing the SLE API; and

b) SLE user and SLE provider applications communicating with each other using the SLE API on both.

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

1.5 DOCUMENT STRUCTURE

1.5.1 ORGANIZATION

This document is organized as follows:

– section 1 provides purpose and scope of this Recommended Practice, identifies conventions, and lists definitions and references used throughout the document;

– section 2 describes the extension of the API model defined in reference [5] to include support for the FSP service;

– section 3 contains detailed specifications for the API components and for applications using the API;

– annex A provides a formal specification of the API interfaces and data types specific to the FSP service;

– annex B lists all acronyms used within this document;

– annex C lists informative references.

1.5.2 SLE SERVICE DOCUMENTATION TREE

The SLE suite of Recommended Standards is based on the cross support model defined in the SLE Reference Model (reference [2]). 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.

CCSDS HISTORICAL DOCUMENT

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.

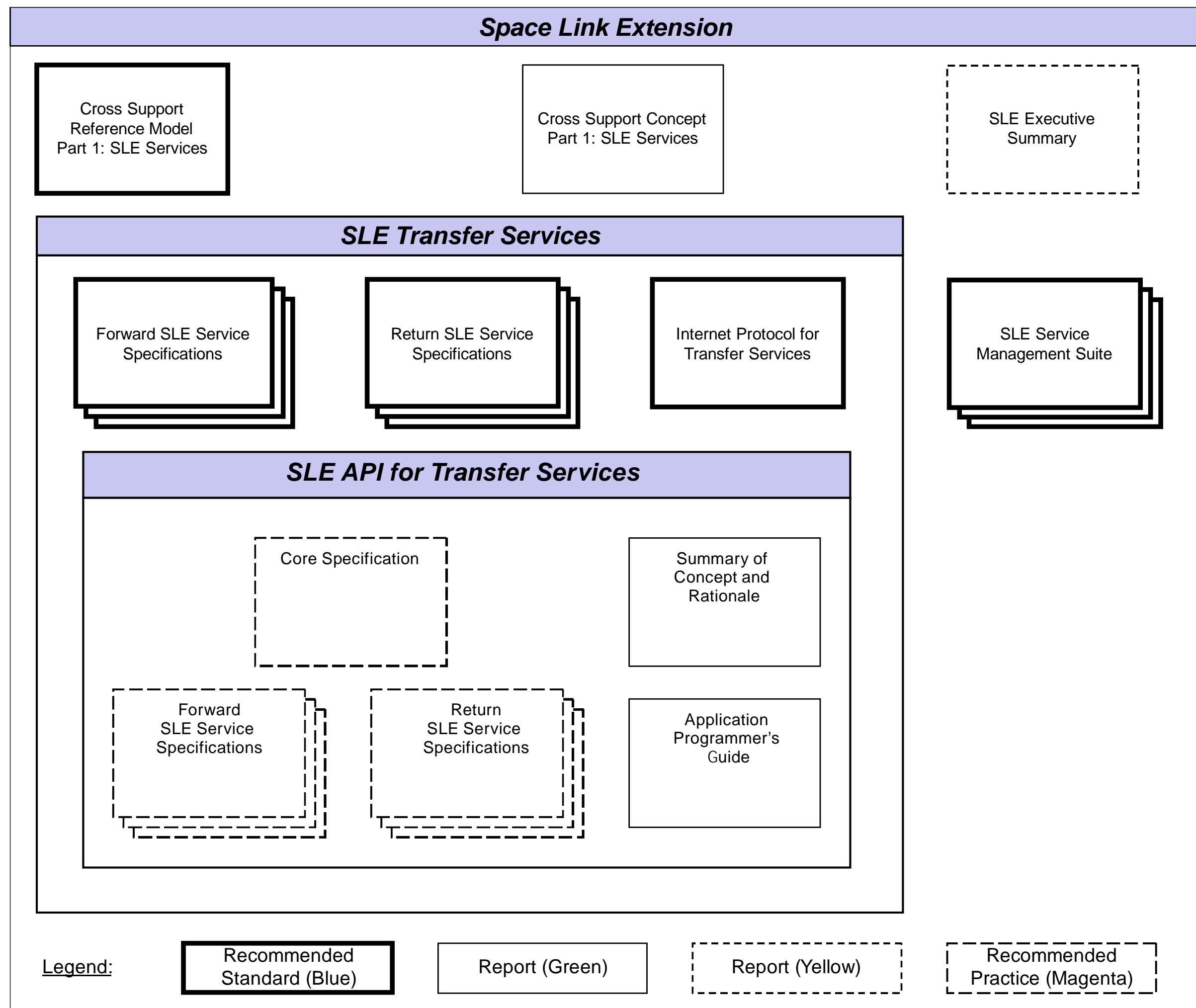

Figure 1-1: SLE Services and SLE API Documentation

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 Services*, a set of Recommended Practice documents specifying the service type-specific extensions of the API for return link SLE services.

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

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

1.6 DEFINITIONS, NOMENCLATURE, AND CONVENTIONS

1.6.1 DEFINITIONS

1.6.1.1 Definitions from TC Space Data Link Protocol

This Recommended Practice makes use of the following terms defined in reference [1]:

a) AD, BD, BC;

b) Command Link Control Word (CLCW);

c) Frame Operation Procedure (FOP);

d) Multiplexer Access Point (MAP);

e) Virtual Channel (VC).

1.6.1.2 Definitions from SLE Reference Model

This Recommended Practice makes use of the following terms defined in reference [2]:

a) Forward Space Packet service;

b) operation;

c) service provider (provider);

d) service user (user);

e) SLE transfer service instance;

f) SLE transfer service production;

g) SLE transfer service provision;

h) space link data unit (SL-DU).

1.6.1.3 Definitions from FSP Service

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

a) association;

b) communications service;

c) confirmed operation;

d) invocation;

e) parameter;

f) performance;

g) port identifier;

h) return;

i) service instance provision period;

j) unconfirmed operation.

1.6.1.4 Definitions from ASN.1 Specification

This Recommended Practice makes use of the following terms defined in reference [6]:

CCSDS HISTORICAL DOCUMENT

a) Object Identifier;

b) Octet String.

1.6.1.5 Definitions from UML Specification

This Recommended Practice makes use of the following terms defined in reference [C7]:

a) Attribute;

b) Base Class;

c) Class;

d) Data Type;

e) Interface;

f) Method.

1.6.1.6 Definitions from API Core Specification

This Recommended Practice makes use of the following terms defined in reference [4]:

a) Application Program Interface;

b) Component.

1.6.2 NOMENCLATURE

The following conventions apply throughout this Recommended Practice:

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

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

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

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

1.6.3 CONVENTIONS

This document applies the conventions defined in reference [4].

The model extensions in section 2 are presented using the Unified Modeling Language (UML) and applying the conventions defined in reference [4].

CCSDS HISTORICAL DOCUMENT

The FSP-specific specifications for API components in section 3 are presented using the conventions specified in reference [4].

The FSP-specific interfaces in annex A are specified using the conventions defined in reference [4].

1.7 REFERENCES

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

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

[1] *TC Space Data Link Protocol*. Recommendation for Space Data Systems Standards, CCSDS 232.0-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, September 2003.

[2] *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.

[3] *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.

[4] *Space Link Extension—Application Program Interface for Transfer Services—Core Specification*. Specification Concerning Space Data System Standards, CCSDS 914.0-M-1. Magenta Book. Issue 1. Washington, D.C.: CCSDS, October 2008.

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

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

2 OVERVIEW

2.1 INTRODUCTION

This section describes the extension of the SLE API model in reference [5] for support of the FSP service. Extensions are needed for the API components API Service Element and SLE Operations.

In addition to the extensions defined in this section, the component API Proxy must support encoding and decoding of FSP-specific protocol data units.

2.2 PACKAGE FSP SERVICE INSTANCES

2.2.1 OVERVIEW

The FSP extensions to the component API Service Element are defined by the package FSP Service Instances. Figure 2-1 provides an overview of this package. The diagram includes classes from the package API Service Element specified in reference [5], which provide applicable specifications for the FSP service.

The package adds two service instance classes:

a) FSP SI User, supporting the service user role; and

b) FSP SI Provider, supporting service provider role.

These classes correspond to the placeholder classes I<SRV>_SI User and I<SRV>_SI Provider defined in reference [5].

Both classes are able to handle the specific FSP operations.

For the class FSP SI User, this is the only extension of the base class SI User.

The class FSP SI Provider adds three new interfaces:

a) `IFSP_SIAdmin` by which the application can set FSP-specific configuration parameters;

b) `IFSP_FOPMonitor` by which the application can initialize and update parameters related to the FOP machine; and

c) `IFSP_SIUpdate` by which the application must update dynamic status information, required for generation of status reports.

These interfaces correspond to the placeholder interfaces `I<SRV>_SIAdmin` and `I<SRV>_SIUpdate` defined in reference [5]. For the FSP service, the conceptual interface `I<SRV>_SIAdmin` is split into the two interfaces `IFSP_SIAdmin` and `IFSP_FOPMonitor` because of the large number of parameters that must be handled.

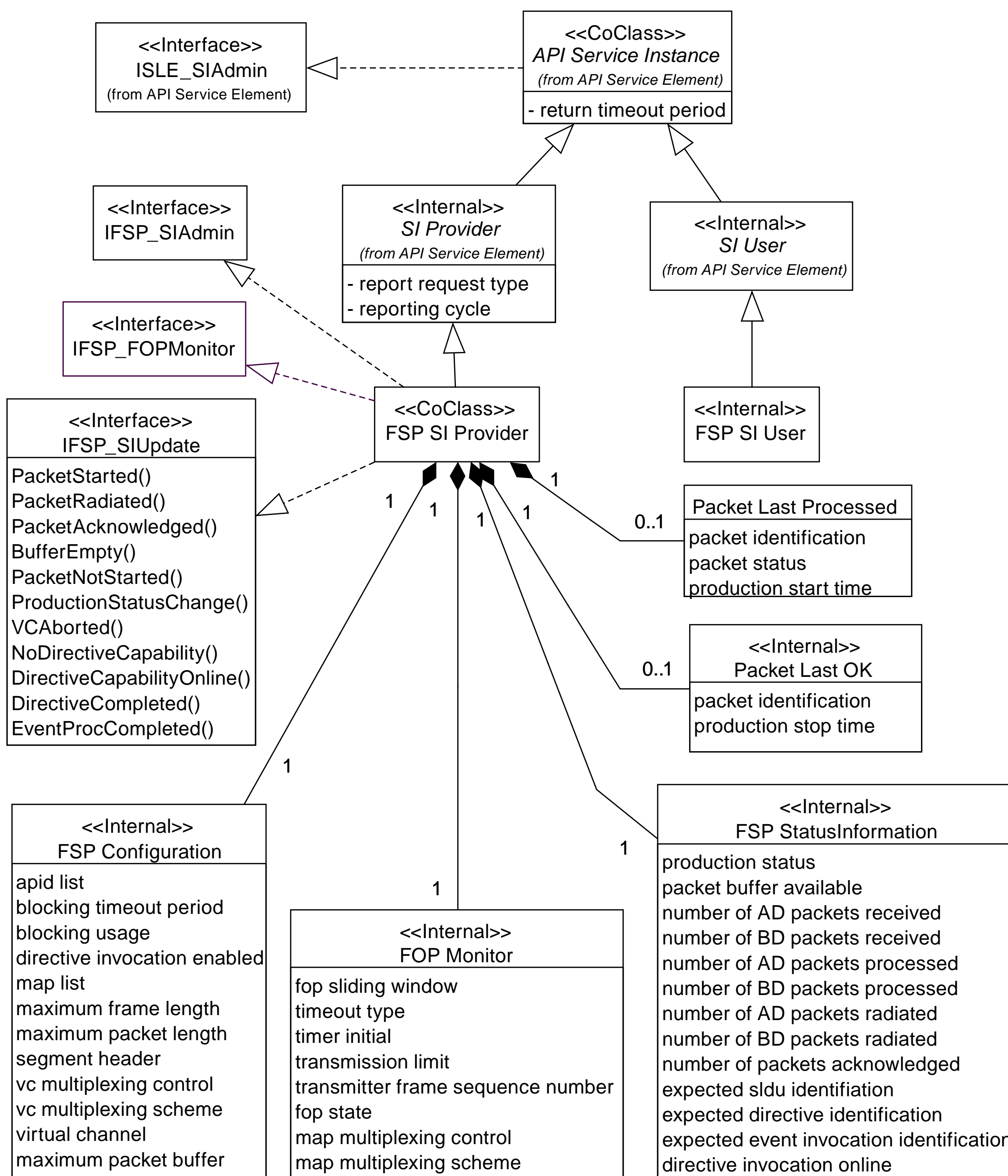

Figure 2-1: FSP Service Instances

FSP-specific service parameters are defined by the internal classes FSP Configuration and FOP Monitor. The class FSP Status Information defines dynamic status parameters maintained by the service instance. In addition, the service instance maintains a set of parameters for the last packet for which processing started and for the last packet for which processing was successfully completed. These parameters are defined by the classes Packet Last Processed and Packet Last OK.

CCSDS HISTORICAL DOCUMENT

All specifications provided in this section refer to a single service instance. If more than one service instance is used, each service instance must be configured and updated independently.

2.2.2 COMPONENT CLASS FSP SI USER

The class defines a FSP service instance supporting the service user role. It ensures that SLE PDUs passed by the application and by the association are supported by the FSP service and handles the FSP operation objects defined in 2.3. It does not add further features to those provided by the base class SI User.

2.2.3 COMPONENT CLASS FSP SI PROVIDER

The class defines a FSP service instance supporting the service provider role. It exports the interfaces `IFSP_SIAdmin` for configuration of the service instance after creation, `IFSP_FOPMonitor` for update of `FOP` parameters, and `IFSP_SIUpdate` for update of dynamic status parameters during operation.

2.2.3.1 Responsibilities

2.2.3.1.1 Service Specific Configuration

The service instance implements the interface `IFSP_SIAdmin` to set the FSP-specific configuration parameters defined by the class FSP Configuration. The methods of this interface must be called after creation of the service instance. When all configuration parameters (including those set via the interface `ISLE_SIAdmin` and the interface `IFSP_FOPMonitor`) have been set, the method `ISLE_SIAdmin::ConfigCompleted()` must be called. This method verifies that all configuration parameter values are defined and are in the range defined in reference [3].

In addition, the interface `IFSP_SIAdmin` provides read access to the configuration parameters.

2.2.3.1.2 Initialization and Update of FOP Parameters

The service instance implements the interface `IFSP_FOPMonitor` for initialization and update of the parameters defined by the class FOP Monitor. The API service instance uses these parameters only to respond to GET-PARAMETER invocations. The application must initialize these parameters when configuring the service instance and update them whenever they change during the lifetime of the service instance.

Changes of the parameter values might occur because of directives invoked by a service user on the same or on a different service instance, because of events detected by the FOP machine, or because of management action. In order to ensure that the service instance always reports the correct parameter value, updates must be forwarded independent of the service instance state.

CCSDS HISTORICAL DOCUMENT

2.2.3.1.3 Update of Dynamic Status Parameters

The class implements the interface IFSP_SIUpdate to inform the service instance of specific events in the FSP production process. The methods of this interface update status parameters defined by the classes FSP Status Information, Packet Last Processed, and Packet Last OK. The events reported via IFSP_SIUpdate and the parameters updated via this interface are listed in table 2-1.

In order to ensure that the status information is always up to date the events listed in table 2-1 must be reported to the service instance during its complete lifetime, independent of the state of the service instance.

In addition, the class derives some of the parameters in FSP Status Information from FSP PDUs exchanged between the service user and the service provider. The methods used to update each of the parameters are defined in 3.1.4.13.

The interface IFSP_SIUpdate provides read access to all status parameters.

2.2.3.1.4 Generation of Notifications

If events reported via the interface IFSP_SIUpdate require that a FSP–ASYNC–NOTIFY invocation be sent to the service user, the class generates and transmits the invocation if that is requested by the application and if the state of the service instance is ‘active’ or ‘ready’. The notifications that are generated and transmitted by the class are listed in table 2-1. The application can opt not to use this feature, but to generate the notification itself and transmit it using the interface ISLE_ServiceInitiate.

2.2.3.1.5 Handling of the FSP–GET-PARAMETER Operation

The class responds autonomously to FSP–GET–PARAMETER invocations. It generates the appropriate FSP–GET–PARAMETER return using the parameters maintained by the classes FSP Configuration, FOP Monitor, and FSP Status Information.

2.2.3.1.6 Status Reporting

The class generates FSP–STATUS–REPORT invocations when required using the parameters maintained by the classes FSP Status Information, Packet Last Processed, and Packet Last OK.

2.2.3.1.7 Processing of FSP Protocol Data Units

The class ensures that SLE PDUs passed by the application and by the association are supported by the FSP service and handles the FSP operation objects defined in 2.3.

Table 2-1: Production Events Reported via the Interface IFSP_SIUpdate

NOTE – The notification type actually transmitted depends on the method arguments and partially or the value of the production status.

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 (See Note 1) number of BD packets processed (See Note 2) packet buffer available	packet processing started
Radiation of a packet completed	`PacketRadiated`	packet id transmission mode radiation end time (See Note 2)	packet id last OK (See Note 2) packet status production stop time(See Note 2) number of AD packets radiated(See Note 1) number of BD packets radiated(See Note 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

Event	Method	Arguments	Status parameters updated	Notification sent
Processing of a packet could not be started because the latest production time expired; the production status was interrupted; The required transmission mode capability 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 (See Note 4) fop alert (See Note 5) available buffer size	production status packet status (See Note 3) packet buffer available	production operational production interrupted production halted transmission mode capability change transmission mode mismatch
The VC was aborted by a directive	`VCAborted`	affected packets list (See Note 4) available buffer size	packet status (See Note 3) production status packet buffer available	VC aborted
The service instance with directive invocation capability is no longer bound	`NoDirectiveCapability`		directive invocation online	no invoke directive capability on this VC
A service instance with directive invocation capability has bound	`DirectiveCapabilityOnline`		directive invocation online	invoke directive capability on this VC established
Processing of a directive completed	`DirectiveCompleted`	directive id result fop alert (See Note 6)		positive confirm response to directive negative confirm response to directive

Event	Method	Arguments	Status parameters updated	Notification sent
Processing of a thrown event completed	`EventProcCompleted`	event invocation id event proc result		action list completed action list not completed event condition evaluated to false

NOTES

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.

2.2.3.1.8 Processing of FSP–TRANSFER–DATA Invocations

For incoming FSP–TRANSFER–DATA invocations the class performs the checks defined in 3.1.5.1 in addition to those defined in reference [4].

In contrast to standard handling of confirmed operations, the service instance is allowed to pass the operation object to the application after setting the correct diagnostic if these checks fail. The application is expected to insert the next expected packet identification and the available buffer size into the operation object and pass it back to service instance via the interface `ISLE_ServiceInitiate`. The reasons for this specification are explained in 2.2.9.3.

2.2.3.1.9 Processing of FSP–THROW-EVENT invocations

In contrast to standard handling of confirmed operations, the service instance is allowed to pass the operation object to the application after setting the correct diagnostic if checks performed by the service element fail. The application is expected to insert the next expected event invocation identifier into the operation object and pass it back to service instance via the interface `ISLE_ServiceInitiate`. The reasons for this specification are explained in 2.2.9.3.

2.2.3.1.10 Processing of FSP–INVOKE-DIRECTIVE invocations

For incoming FSP–INVOKE-DIRECTIVE invocations the class verifies that the service instance has the capability to invoke directives in addition to the checks defined in reference [5].

In contrast to standard handling of confirmed operations, the service instance is allowed to pass the operation object to the application after setting the correct diagnostic if checks performed by the service element fail. The application is expected to insert the next expected directive invocation identifier into the operation object and pass it back to service instance via the interface `ISLE_ServiceInitiate`. The reasons for this specification are explained in 2.2.9.3.

2.2.4 INTERNAL CLASS FSP CONFIGURATION

The class defines the configuration parameters that can be set via the interface `IFSP_SIAdmin`. These parameters are defined by reference [3]. Table 2-2 describes how the service instance uses these parameters. The column labeled ‘Upd’ indicates whether an update of these parameters is allowed after the initial configuration has been completed. Table 2-2 only indicates which parameters must not be modified in order to ensure proper operation of the API. Updates allowed by the table might be inhibited because of other constraints.

Table 2-2: FSP Configuration Parameters Handled by the Service Element

Parameter	Used for	Upd
`apid-list`	FSP–GET–PARAMETER	Y
`blocking-timeout-period`	FSP–GET–PARAMETER	Y
`blocking-usage`	FSP–GET–PARAMETER	Y
`directive-invocation-enabled`	FSP–GET–PARAMETER Checking of FSP-INVOKE-DIRECTIVE	Y
`map-list`	FSP–GET–PARAMETER Checking of FSP-TRANSFER-DATA	Y
`maximum-frame-length`	FSP–GET–PARAMETER	Y
`maximum-packet-length`	FSP–GET–PARAMETER Checking of FSP-TRANSFER-DATA	Y
`segment-header`	FSP–GET–PARAMETER	Y
`vc-multiplexing-control`	FSP–GET–PARAMETER	Y
`vc-multiplexing-scheme`	FSP–GET–PARAMETER	Y
`virtual-channel`	FSP–GET–PARAMETER	Y
`maximum-packet-buffer-size`	Value of the status parameter packet buffer available after configuration, FSP-STOP, FSP-PEER-ABORT, and protocol abort	N

2.2.5 INTERNAL CLASS FOP MONITOR

The class defines the FOP parameters that can be initialized and updated via the interface `IFSP_FOPMonitor`. These parameters are defined by reference [3]. Table 2-3 describes how the service element uses these parameters. The parameters 'map multiplexing scheme' and 'map multiplexing control' are assigned to this class because 'map multiplexing control' can be modified by a service user via a directive in the same way as FOP control parameters.

Table 2-3: FSP FOP Parameters Handled by the Service Element

Parameter	Used for
`fop-sliding-window`	FSP–GET–PARAMETER
`timeout-type`	FSP–GET–PARAMETER
`timer-initial`	FSP–GET–PARAMETER
`transmission-limit`	FSP–GET–PARAMETER
`transmitter-frame-sequence-number`	FSP–GET–PARAMETER
`fop-state`	FSP–GET–PARAMETER
`map-multiplexing-control`	FSP–GET–PARAMETER
`map-multiplexing-scheme`	FSP–GET–PARAMETER

2.2.6 INTERNAL CLASS FSP STATUS INFORMATION

The class defines status parameters handled by the service instance. The parameters are defined by reference [3]. Table 2-4 describes how the service element updates each of the parameters and how it uses the parameters.

Table 2-4: FSP Status Parameters Handled by the Service Element

Parameter	Update	Used for
`production-status`	– set by methods of `IFSP_SIUpdate`	status reports notifications
`packet-buffer-available`	– set to maximum at configuration time – set by methods of `IFSP_SIUpdate` – extracted from FSP-TRANSFER-DATA returns – reset to maximum following a notification 'buffer empty' – reset to maximum following FSP–STOP, FSP–PEER–ABORT and protocol abort	status reports notifications
`number-of-AD-packets-received`	– set to zero at configuration time – incremented for every FSP-TRANSFER-DATA return with transmission mode 'sequence controlled' and a positive result	status reports
`number-of-BD-packets-received`	– set to zero at configuration time – incremented for every FSP-TRANSFER-DATA return with transmission mode 'expedited' and a positive result	status reports
`number-of-AD-packets-processed`	– set to zero at configuration time – incremented with every call to `PacketStarted()` and `PacketNotStarted()` with the argument transmission mode set to 'sequence controlled'	status reports
`number-of-BD-packets-processed`	– set to zero at configuration time – incremented with every call to `PacketStarted()` and `PacketNotStarted()` with the argument transmission mode set to 'expedited'	status reports
`number-of-AD-packets-radiated`	– set to zero at configuration time – incremented with every call to `PacketRadiated()` with the argument transmission mode set to 'sequence controlled'	status reports
`number-of-BD-packets-radiated`	– set to zero at configuration time – incremented with every call to `PacketRadiated()` with the argument transmission mode set to 'expedited'	status reports
`number-of-packets-acknowledged`	– set to zero at configuration time – incremented with every call to `PacketAcknowledged()`	status reports

Parameter	Update	Used for
expected-sldu-identification	– set to zero at configuration time – copied from the first packet identification parameter of FSP-START invocations if the application transmits a return with a positive result. – copied from packet identification of FSP-TRANSFER-DATA returns	FSP-GET-PARAMETER
expected-directive-identification	– set to zero at configuration time – extracted from FSP-INVOKE-DIRECTIVE returns	FSP-GET-PARAMETER
expected-event-invocation-identification	– set to zero at configuration time – extracted from FSP-THROW-EVENT returns	FSP-GET-PARAMETER
directive-invocation-online	See Note	FSP-GET-PARAMETER

NOTE – The value of the parameter directive-invocation-online at the time the service instance is configured must be specified by the application if directive invocation is not enabled for the service instance. Subsequently, the API updates the parameter when the methods NoDirectiveCapability() and DirectiveCapabilityOnline() are invoked on the interface IFSP_SIUpdate. For service instances with invocation capability enabled, the parameter is initialized to 'no' and subsequently set to 'yes' when the user binds and to 'no' when the user unbinds or the association is aborted.

2.2.7 INTERNAL CLASS PACKET LAST PROCESSED

The class defines the parameters maintained by the service instance for the last packet for which processing started or was attempted. These parameters are defined in reference [3].

All parameters are set via methods in the interface IFSP_SIUpdate (see table 2-1) and are used in status reports and notifications.

2.2.8 INTERNAL CLASS PACKET LAST OK

The class defines the parameters maintained by the service instance for the last packet for which processing was completed. For BD packets completion implies that the packet was radiated. AD packets are considered complete when all segments of the packet have been acknowledged by a CLCW.

These parameters are defined in reference [3]. All parameters are set via methods in the interface IFSP_SIUpdate (see table 2-1) and are used in status reports and notifications.

2.2.9 FEATURES NOT HANDLED BY THE PROVIDER SIDE SERVICE INSTANCE

2.2.9.1 Introduction

As a general approach, this specification only states what the API is required to do. Features not identified in this specification cannot be expected from a conforming implementation. This subsection deviates from this approach by discussing features not provided by the API, with the intention to clarify the borderline between the application and the API Service Element.

In addition, this subsection outlines the rationale for the distribution of responsibilities between the application and the API Service Element in this specification.

2.2.9.2 Production Status

Reference [3] defines the parameter production status, which reflects the state of the FSP production engine. The value of the production status is not only included in status reports and notifications, but also determines whether invocations of the operations FSP–BIND and FSP–START can be accepted or not. The production status also has an impact on processing of FSP–TRANSFER–DATA operations, which is discussed in 2.2.9.4.

Table 2-5 lists the possible values of the production status and the required processing of BIND and START invocations.

Table 2-5: FSP Production Status

Production Status	BIND invocation	START invocation
halted	reject (out of service)	reject (out of service)
configured	accept	accept
operational	accept	accept
interrupted	accept	reject (unable to comply)

In a multi-threaded environment, the value of the production status can change concurrently with processing within the service element. That implies that the value can change after a PDU has been processed by the service element but before the same PDU is handled by the application. Because the service element cannot guarantee that the result of a test is still valid when the PDU reaches the application, this specification does not require that the service element check the production status.

This specification does not exclude that implementations of the service element check the production status and reject BIND or START invocations if required. If both the API and the application are single-threaded, the application could rely on such checks. However, applications cannot expect that other implementations provide the same service. Therefore, applications wishing to maintain substitutability of API components should not rely on such behavior.

2.2.9.3 TRANSFER-DATA, INVOKE DIRECTIVE, and THROW-EVENT

For FSP–TRANSFER–DATA returns, reference [3] requires that the provider inserts the next expected packet identification and the available packet buffer size. For FSP-INVOKE-DIRECTIVE and FSP-THROW-EVENT returns, reference [3] requires that the provider inserts the next expected directive identification or event invocation identification, respectively. These parameters are available to the service element via the procedures described in 2.2.6. However, the following must be considered.

A service user is not required to wait for a FSP–TRANSFER–DATA return before transmitting the next FSP–TRANSFER–DATA invocation. Therefore, several FSP–TRANSFER–DATA invocations can be in transit. Depending on the implementation of the service element and of the provider application, FSP–TRANSFER–DATA invocations might be queued between the service element and the application. In such a case, the service element cannot know what values to insert for the next packet identification and the available buffer size when it needs to generate a FSP–TRANSFER–DATA return with a negative result. The same considerations apply to the FSP-INVOKE-DIRECTIVE and FSP–THROW–EVENT operations.

Therefore, this Recommended Practice defines a procedure for the operations FSP–TRANSFER–DATA, FSP-INVOKE-DIRECTIVE, and FSP–THROW–EVENT, which deviates from the standard approach described in reference [5]. When a check performed by the service element fails, the service element may set the appropriate diagnostic in the operation object and pass the operation object to the application. The application is expected to check the result of an invocation. If the result is negative, the application shall insert the next expected packet identification and the available buffer size, the next expected directive identification, or the next expected event invocation identification into the operation object and then pass it back to the service element using the method `InitiateOpReturn()` in the interface `ISLE_ServiceInitiate`.

This specification does not exclude that implementations generate a FSP–TRANSFER–DATA return, a FSP-INVOKE-DIRECTIVE return, or a FSP–THROW–EVENT return if it is possible to insert the correct values for the return parameters. An implementation can apply any of the following approaches:

a) an implementation can always pass invocations for which a check has failed to the application;

b) an implementation can prevent queuing of invocations by withholding an invocation until the previous invocation has been confirmed by the application. In that case, it can always generate the appropriate return when needed; or

c) an implementation can decide to pass invocations to the application on a case-by-case basis.

Applications wishing to maintain substitutability of API components should always expect to receive FSP–TRANSFER–DATA invocations, FSP-INVOKE-DIRECTIVE invocations, and FSP–THROW–EVENT invocations with a negative result from the service element.

2.2.9.4 Processing of TRANSFER-DATA Invocations

2.2.9.4.1 Blocked State of the Service Instance

When a packet cannot be processed because the production status becomes non-operational or because the latest production start time expired the service instance becomes blocked and further FSP–TRANSFER–DATA invocations must be rejected. In order to clear the situation, the service user must invoke a FSP–STOP operation followed by a FSP–START operation.

The event causing the blocked state of the service instance can depend on the production status, which can change concurrently with processing in the service element. In a multi-threaded environment, the service element cannot guarantee that a FSP–TRANSFER–DATA invocation that passed the test of the blocked state is still valid when it reaches the application. Therefore, this specification does not require the service element to perform that check.

This specification does not exclude that implementations check the blocked state of the service instance. If both the API and the application are single-threaded, the application could rely on such checks. However, applications cannot expect that other implementations provide the same service. Applications wishing to maintain substitutability of API components must not rely on such behavior.

2.2.9.4.2 Transmission Mode Capability

When an AD packet cannot be processed because the required transmission mode capability is not available the AD mode becomes blocked and further AD packets must be rejected. In order to clear the situation, the user must invoke an FSP-TRANSFER-DATA operation with the transmission mode ‘sequence controlled and unblock AD’.

The transmission mode capability can change and the AD service can become blocked concurrently with processing in the service element. In a multi-threaded environment, the service element cannot guarantee that a FSP–TRANSFER–DATA invocation that passed the test of the AD blocked state is still acceptable when it reaches the application. Therefore, this specification does not require the service element to perform that check.

This specification does not exclude that implementations check whether the AD service is blocked. If both the API and the application are single-threaded, the application could rely on such checks. However, applications cannot expect that other implementations provide the same service. Applications wishing to maintain substitutability of API components must not rely on such behavior.

2.2.9.4.3 Checking of Time Parameters

FSP–TRANSFER–DATA invocations carry parameters that specify the earliest and latest production start times. Reference [3] requires the service provider to check that these times are not expired at the time the invocation reaches the provider. It cannot be excluded that such a time expires after the invocation has been processed by the service element, but before it reaches the application. Therefore, this specification does not require the service element

to perform these checks. The service element is, however, required to verify that time periods are defined in a consistent manner.

This Recommended Practice does not exclude that implementations check times against current time. However, applications wishing to maintain substitutability of API components must not rely on such behavior.

2.2.9.4.4 APID, Packet Version, and Packet Type

Reference [3] requires that the FSP service provider verify that:

a) the APID of a packet matches one of the entries in the list of permitted APIDs;

b) the packet version is supported by CCSDS and the service instance; and

c) the packet type specifies telecommand.

For these checks it is necessary to inspect the packet delivered by the parameter ‘data’ in the TRANSFER-DATA operation. Because the SLE Application Program Interface generally handles space link data units as opaque data types, this specification does not require a service element to perform these checks.

This specification does not exclude that implementations check the APID, packet version, and packet type. However, applications wishing to maintain substitutability of API components must not rely on such behavior.

2.2.9.5 Production Time

Reference [3] defines a production period, i.e., the period in which the FSP production engine is able to process packets. This period must overlap with the scheduled provision period of the service instance but need not be the same. Reference [3] requires the service provider to check the validity of FSP–START invocations and FSP–TRANSFER–DATA invocations against the production period.

This specification does not require a service element to perform these checks, as they are related to service production and not to service provisioning.

2.3 PACKAGE FSP OPERATIONS

Figure 2-2 shows the operation object interfaces required for the FSP service. The package FSP Operations adds operation objects for the following FSP operations:

a) FSP–START;

b) FSP–TRANSFER–DATA;

c) FSP–ASYNC–NOTIFY;

d) FSP–STATUS–REPORT;

e) FSP–GET–PARAMETER;

f) FSP–THROW–EVENT;

g) FSP–INVOKE–DIRECTIVE.

For other operations the API uses the common operation objects defined in reference [5].

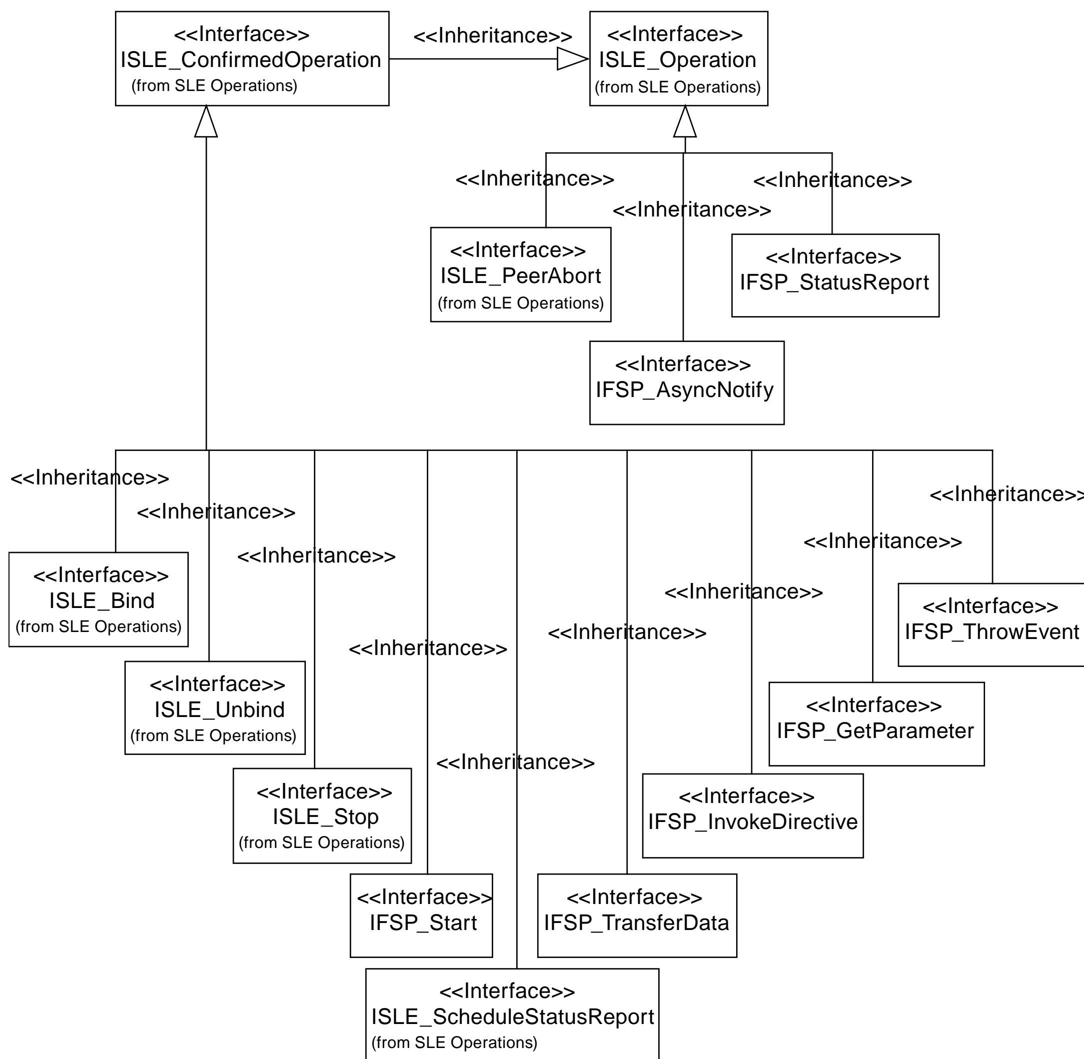

Figure 2-2: FSP Operation Objects

Table 2-6 maps FSP operations to operation object interfaces.

Table 2-6: Mapping of FSP Operations to Operation Object Interfaces

FSP Operation	Operation Object Interface	Defined in Package
FSP–BIND	`ISLE_Bind`	SLE Operations
FSP–UNBIND	`ISLE_Unbind`	SLE Operations
FSP–START	`IFSP_Start`	FSP Operations
FSP–STOP	`ISLE_Stop`	SLE Operations
FSP–TRANSFER–DATA	`IFSP_TransferData`	FSP Operations
FSP–ASYNC–NOTIFY	`IFSP_AsyncNotify`	FSP Operations
FSP–SCHEDULE–STATUS–REPORT	`ISLE_ScheduleStatusReport`	SLE Operations
FSP–STATUS–REPORT	`IFSP_StatusReport`	FSP Operations
FSP–GET–PARAMETER	`IFSP_GetParameter`	FSP Operations
FSP–THROW–EVENT	`IFSP_ThrowEvent`	FSP Operations
FSP–INVOKE–DIRECTIVE	`IFSP_InvokeDirective`	FSP Operations
FSP–PEER–ABORT	`ISLE_PeerAbort`	SLE Operations

2.4 SECURITY ASPECTS OF THE SLE FORWARD SPACE PACKET (FSP) TRANSFER SERVICE

2.4.1 SECURITY BACKGROUND/INTRODUCTION

The SLE transfer services explicitly provide authentication and access control. Additional security capabilities, if required, are levied on the underlying communication services that support the SLE transfer services. The SLE transfer services are defined as layered application services operating over underlying communication services that must meet certain requirements but which are otherwise unspecified. Selection of the underlying communication services over which real SLE implementations connect is based on the requirements of the communicating parties and/or the availability of CCSDS-standard communication technology profiles and proxy specifications. Different underlying communication technology profiles are intended to address not only different performance requirements but also different security requirements. Missions and service providers are expected to select from these technology profiles to acquire the performance and security capabilities appropriate to the mission. Specification of the various underlying communication technologies, and in particular their associated security provisions, are outside the scope of this Recommendation.

The SLE FSP transfer service transfers data that is destined for a mission spacecraft. As such, the SLE FSP transfer service has custody of the data for only a portion of the end-to-end data path between MDOS and mission spacecraft. Consequently the ability of an SLE transfer service to secure the transfer of mission spacecraft data is limited to that portion of the end-to-end path that is provided by the SLE transfer service (i.e., the terrestrial link between the MDOS and the ground termination of the ground-space link to the mission spacecraft). End-to-end security must also involve securing the data as it crosses the ground-space link, which can be provided by some combination of securing the mission data itself (e.g., encryption of the mission data within CCSDS space packets) and securing the ground-space link (e.g., encryption of the physical ground-space link). Thus while the SLE FSP transfer service plays a role in the end-to-end security of the data path, it does not control and cannot ensure that end-to-end security. This component perspective is reflected in the security provisions of the SLE transfer services.

2.4.2 STATEMENTS OF SECURITY CONCERNS

This section identifies SLE FSP transfer service support for capabilities that responds to these security concerns in the areas of data privacy, data integrity, authentication, access control, availability of resources, and auditing.

2.4.2.1 Data Privacy (Also Known As Confidentiality)

This SLE FSP transfer service specification does not define explicit data privacy requirements or capabilities to ensure data privacy. Data privacy is expected to be ensured outside of the SLE transfer service layer, by the mission application processes that communicate over the SLE transfer service, in the underlying communication service that lies under the SLE transfer service, or some combination of both. For example, mission application processes might apply end-to-end encryption to the contents of the CCSDS space link data units carried as data by the SLE transfer service. Alternatively or in addition, the network connection between the SLE entities might be encrypted to provide data privacy in the underlying communication network.

2.4.2.2 Data Integrity

The SLE FSP service requires that each transferred space packet be accompanied by a sequence number, which must increase monotonically. Failure of a space packet to be accompanied by the expected sequence number causes the space packet to be rejected (see 3.6.2.18.1 d) in reference [3]). This constrains the ability of a third party to inject additional command data into an active FSP transfer service instance.

The SLE FSP transfer service defines and enforces a strict sequence of operations that constrain the ability of a third party to inject operation invocations or returns into the transfer service association between a service user and provider (see 4.2.2 in reference [3]). This

constrains the ability of a third party to seize control of an active FSP transfer service instance without detection.

The SLE FSP transfer service requires that the underlying communication service transfer data in sequence, completely and with integrity, without duplication, with flow control that notifies the application layer in the event of congestion, and with notification to the application layer in the event that communication between the service user and the service provider is disrupted (see 1.3.1 in reference [3]). No specific mechanisms are identified, as they will be an integral part of the underlying communication service.

2.4.2.3 Authentication

This SLE FSP transfer service specification defines authentication requirements (see 3.1.5 in reference [3]), and defines `initiator-identifier`, `responder-identifier`, `invoker-credentials`, and `performer-credentials` parameters of the service operation invocations and returns that are used to perform SLE transfer service authentication. The procedure by which SLE transfer service operation invocations and returns are authenticated is described in annex F of the Cross Support Service Green Book (reference [C2]). The SLE transfer service authentication capability can be selectively set to authenticate at one of three levels: authenticate every invocation and return, authenticate only the BIND operation invocation and return, or perform no authentication. Depending upon the inherent authentication available from the underlying communication network, the security environment in which the SLE service user and provider are operating, and the security requirements of the spaceflight mission, the SLE transfer service authentication level can be adapted by choosing the SLE operation invocations and returns that shall be authenticated. Furthermore, the mechanism used for generating and checking the credentials and thus the level of protection against masquerading (simple or strong authentication) can be selected in accordance with the results of a threat analysis.

2.4.2.4 Access Control

This SLE FSP transfer service specification defines access control requirements (see 3.1.4 in reference [3]), and defines `initiator-identifier` and `responder-identifier` parameters of the service operation invocations and returns that are used to perform SLE transfer service access control. The procedure by which access to SLE transfer services is controlled is described in annex F of the Cross Support Service Green Book (reference [C2]).

2.4.2.5 Availability of Resources

The SLE transfer services are provided via communication networks that have some limit to the resources available to support those SLE transfer services. If these resources can be diverted from their support of the SLE transfer services (in what is commonly known as “denial of service”) then the performance of the SLE transfer services may be curtailed or inhibited. This SLE FSP transfer service specification does not define explicit capabilities to

prevent denial of service. Resource availability is expected to be ensured by appropriate capabilities in the underlying communication service. The specific capabilities will be dependent upon the technologies used in the underlying communication service and the security environment in which the transfer service user and provider operate.

2.4.2.6 Auditing

This SLE FSP transfer service specification does not define explicit security auditing requirements or capabilities. Security auditing is expected to be negotiated and implemented bilaterally between the spaceflight mission and the service provider.

2.4.3 POTENTIAL THREATS AND ATTACK SCENARIOS

The SLE FSP transfer service depends on unspecified mechanisms operating above the SLE transfer service (between a mission spacecraft application process and its peer application process on the ground), underneath the SLE transfer service in the underlying communication service, or some combination of both, to ensure data privacy (confidentiality). If no such mechanisms are actually implemented, or the mechanisms selected are inadequate or inappropriate to the network environment in which the mission is operating, an attacker could read the command data contained in the FSP protocol data units as they traverse the WAN between service user and service provider.

The SLE FSP transfer service constrains the ability of a third party to seize control of an active SLE transfer service instance, or to inject extra command data into a service instance, but it does not specify mechanisms that would prevent an attacker from intercepting the protocol data units and replacing the contents of the `data` parameter. The prevention of such a replacement attack depends on unspecified mechanisms operating above the SLE transfer service (between a mission spacecraft application process and its peer application process on the ground), underneath the SLE transfer service in the underlying communication service, in bilaterally-agreed extra capabilities applied to the SLE transfer service (e.g., encryption of the `data` parameter) or some combination of the three. If no such mechanisms are actually implemented, or the mechanisms selected are inadequate or inappropriate to the network environment in which the mission is operating, an attacker could “hijack” an established SLE FSP transfer service instance and overwrite the commands in the protocol data units to subvert or destroy the operation of the spacecraft.

If the SLE transfer service authentication capability is not used and if authentication is not ensured by the underlying communication service, attackers may somehow obtain valid `initiator-identifier` values and use them to initiate SLE transfer service instances by which they could subvert or destroy the mission.

The SLE FSP transfer service depends on unspecified mechanisms operating in the underlying communication service to ensure that the supporting network has sufficient resources to provide sufficient support to legitimate users. If no such mechanisms are actually implemented, or the mechanisms selected are inadequate or inappropriate to the

network environment in which the mission is operating, an attacker could prevent legitimate users from communicating with their spacecraft, causing degradation or even loss of the mission.

If the provider of SLE FSP transfers service provides no security auditing capabilities, or if a user chooses not to employ auditing capabilities that do exist, then attackers may delay or escape detection long enough to do serious (or increasingly serious) harm to the mission.

2.4.4 CONSEQUENCES OF NOT APPLYING SECURITY

The consequences of not applying security to the SLE FSP transfer service are possible degradation and loss of ability to command the spacecraft, and even loss of the spacecraft itself.

3 FSP SPECIFIC SPECIFICATIONS FOR API COMPONENTS

3.1 API SERVICE ELEMENT

3.1.1 SERVICE INSTANCE CREATION

3.1.1.1 The service element shall allow creation of service instances supporting the Forward Space Packet (FSP) service.

3.1.1.2 FSP service instances shall be provided to support the service provider role and the service user role.

NOTE – As specified a by reference [5], a given implementation of the component API Service Element might support only the user role, only the provider role, or both roles.

3.1.2 SERVICE INSTANCE CONFIGURATION

3.1.2.1 The service element shall provide the interface `IFSP_SIAdmin` for configuration of a provider-side service instance after creation.

3.1.2.2 The interface `IFSP_SIAdmin` shall provide methods to set the following configuration parameters, which the service element uses to respond to GET-PARAMETER invocations received from the service user:

a) `apid-list`;

b) `blocking-timeout-period`;

c) `blocking-usage`;

d) `directive-invocation-enabled`;

e) `map-list`;

f) `maximum-frame-length`;

g) `maximum-packet-length`;

h) `segment-header`;

i) `vc-multiplexing-control`;

j) `vc-multiplexing-scheme`; and

k) `virtual-channel`.

NOTE – These parameters are defined in reference [3] for the operation FSP-GET-PARAMETER.

3.1.2.3 The interface `IFSP_SIAdmin` shall provide a method to set the initial value of the parameter `directive-invocation-online` if the configuration parameter `directive-invocation-enabled` is set to 'no'.

NOTE – The parameter `directive-invocation-online` is defined in reference [3] for the operation FSP-GET-PARAMETER. Further processing of this parameter is described in 3.1.4.13.

3.1.2.4 The interface `IFSP_SIAdmin` shall provide methods to set the following parameters, which the service instance uses to initialize parameters of the status report:

a) the maximum size of the packet buffer shall be used to initialize the parameter `packet-buffer-available`;

b) the value of the production status at the time the service instance is configured.

NOTE – Further configuration parameters must be set using the interface `ISLE_SIAdmin` specified in reference [3] and the interface `IFSP_FOPMonitor` defined in 3.1.4.

3.1.2.5 All configuration parameters must be set before the method `ConfigCompleted()` of the interface `ISLE_SIAdmin` is called. If one of the parameters is omitted or the value of a parameter is not within the range specified by reference [3], the method `ConfigCompleted()` shall return an error.

NOTES

1 As defined in reference [5], the service element starts processing of the service instance only after successful configuration.

2 The range of specific parameter values might be further constrained by service management. The service element shall only check on the limits specified by reference [3].

3.1.2.6 Configuration parameters listed in 3.1.2.2 can be modified at any time during operation of the service instance. The service element shall always use the most recent value.

NOTE – Modification of these parameters during the scheduled provision period of the service instance might be inhibited by service management. Such constraints must be handled by the application.

3.1.2.7 Configuration parameters defined in 3.1.2.3 and 3.1.2.4 must not be modified after successful return of the method `ConfigCompleted()` defined in the interface `ISLE_SIAdmin`. The effect of an attempt to set these parameters after completion of the configuration is undefined.

3.1.2.8 The values of the configuration parameters identified in 3.1.2.2 shall remain unmodified following an FSP-UNBIND or FSP-PEER-ABORT operation and following a protocol-abort.

3.1.2.9 The interface `IFSP_SIAdmin` shall provide methods to retrieve the values of the configuration parameters. The values returned by these methods before configuration has been completed are undefined.

3.1.3 INITIALIZATION AND UPDATE OF FOP PARAMETERS

3.1.3.1 The service element shall provide the interface `IFSP_FOPMonitor` for initialization and update of parameters related to the FOP machine in a provider side service instance.

3.1.3.2 The interface `IFSP_FOPMonitor` shall provide methods to set the following parameters, which the service element uses to respond to GET-PARAMETER invocations received from the service user:

a) `fop-sliding-window`;

b) `timeout-type`;

c) `timer-initial`;

d) `transmission-limit`;

e) `transmitter-frame-sequence-number`;

f) `fop-state`;

g) `map-multiplexing-control`; and

h) `map-multiplexing-scheme`.

NOTES

1 These parameters are defined in reference [3] for the operation FSP-GET-PARAMETER.

2 The parameters `map-multiplexing-scheme` and `map-multiplexing-control` are included in this interface because `map-multiplexing-control` can be modified by the service user via a directive in the same way as FOP control parameters.

3.1.3.3 Initial values for the parameters listed in 3.1.3.2 must be set before the method `ConfigCompleted()` of the interface `ISLE_SIAdmin` is called. If one of the parameters is omitted, or the value of a parameter is not within the range specified by reference [3], the method `ConfigCompleted()` shall return an error.

3.1.3.4 During the complete lifetime of the service instance the parameters listed in 3.1.3.2 must be updated via the interface `IFSP_FOPMonitor`, whenever their value changes.

NOTES

1 Changes might occur because of directives invoked by a service user on the same or on a different service instance, because of events detected by the FOP machine, or because of management action.

2 In order to ensure that the service instance always reports the correct parameter value, updates must be reported independent of the service instance state.

3.1.3.5 The interface `IFSP_FOPMonitor` shall provide methods to retrieve the values of the parameters. The values returned by these methods before configuration has been completed are undefined.

3.1.4 STATUS INFORMATION

3.1.4.1 Status Parameters

3.1.4.1.1 The service element shall maintain status parameters for every service instance and use them for generation of status reports, notifications, and for FSP–GET–PARAMETER returns.

NOTES

1 The parameters are defined in reference [3] for the operations FSP-ASYNC-NOTIFY, FSP-STATUS-REPORT, and FSP-GET-PARAMETER.

2 Handling of the parameter `reporting-cycle` defined for the FSP-GET-PARAMETER operation shall be specified in reference [5].

3.1.4.1.2 The service element shall update the following status parameters in the methods of the interface `IFSP_SIUpdate` described in 3.1.4.2:

a) `packet-identification-last-processed`;

b) `packet-status`;

c) `production-start-time`;

d) `packet-identification-last-OK`;

e) `production-stop-time`;

f) `production-status`;

g) `number-of-AD-packets-processed`;

h) `number-of-BD-packets-processed`;

i) `number-of-AD-packets-radiated`;

j) `number-of-BD-packets-radiated`; and

k) `number-of-AD-packets-acknowledged`.

NOTE – The initial values of these parameters following configuration of the service instance are defined in A4.3.

3.1.4.1.3 The service element shall handle the parameter `directive-invocation-online` as defined by the following specifications:

NOTE – The parameter `directive-invocation-online` can be requested by an FSP-GET-PARAMETER invocation.

a) when the parameter `directive-invocation-enabled` is 'no', the initial value shall be set by configuration of the service instance. Subsequently the value shall be set to 'yes' when the method `DirectiveCapabilityOnline()` is invoked and it shall be set to 'no' when the method `NoDirectiveCapability()` is called on the interface `ISLE_SIUpdate`;

b) when the parameter `directive-invocation-enabled` is 'yes', the initial value shall be set to 'no'. Subsequently the value shall be set to 'yes' when the user binds and it shall be reset to 'no' when the user unbinds or when the association is aborted.

3.1.4.1.4 The service element shall handle the parameter `expected-sldu-identification` as defined by the following specifications:

NOTE – The parameter `expected-sldu-identification` can be requested by an FSP-GET-PARAMETER invocation.

a) the parameter shall be set to zero when the service instance is configured;

b) when the application transmits an FSP-START return with a positive result, the value shall be set to the value of the parameter `first-packet-identification` in the FSP-START invocation;

c) the value shall be copied from the parameter `packet-identification` in every FSP-TRANSFER-DATA return issued by the application.

3.1.4.1.5 The service element shall handle the parameter `expected-directive-identification` as defined by the following specifications:

NOTE – The parameter `expected-directive-invocation-identifier` can be requested by an FSP-GET-PARAMETER invocation.

a) the parameter shall be set to zero when the service instance is configured;

b) the value shall be copied from every FSP-INVOKE-DIRECTIVE return issued by the application.

3.1.4.1.6 The service element shall handle the parameter `expected-event-invocation-identifier` as defined by the following specifications:

NOTE – The parameter `expected-event-invocation-identifier` can be requested by an FSP-GET-PARAMETER invocation.

a) the parameter shall be set to zero when the service instance is configured;

b) the value shall be copied from every FSP-THROW–EVENT return issued by the application.

3.1.4.1.7 The service element shall handle the parameter `packet-buffer-available` as defined by the following specifications:

a) at configuration time, the value shall be copied from the configuration parameter `maximum-packet-buffer`, defined in 3.1.2.4;

b) when the application transmits an FSP-TRANSFER-DATA return the value shall be copied from the parameter set by the application;

c) the value shall be updated whenever passed as argument by one of the methods in the interface `IFSP_SIUpdate`;

d) when the application transmits an FSP-STOP return with a positive result, the value shall be copied from the configuration parameter `maximum-packet-buffer`;

e) when the application or the service element transmits an FSP-ASYNC-NOTIFY invocation with the parameter `notification-type` set to 'buffer empty', the value shall be copied from the configuration parameter `maximum-packet-buffer`;

NOTE – The service element shall transmit the notification when requested by the application via the interface `IFSP_SIUpdate` specified in 3.1.4.2.

f) following an FSP-PEER-ABORT operation and following protocol-abort, the value shall be copied from the configuration parameter `maximum-packet-buffer`.

3.1.4.1.8 The service element shall handle the parameter `number-of-AD-packets-received` as defined by the following specifications:

a) the parameter shall be set to zero when the service instance is configured;

b) the parameter shall be incremented whenever the application transmits an FSP-TRANSFER-DATA return with a positive result and the parameter `transmission-mode` was set to 'sequence controlled' in the invocation.

3.1.4.1.9 The service element shall handle the parameter `number-of-BD-packets-received` as follows:

a) the parameter shall be set to zero when the service instance is configured;

b) the parameter shall be incremented whenever the application transmits an FSP-TRANSFER-DATA return with a positive result and the parameter `transmission-mode` was set to 'expedited' in the invocation.

3.1.4.1.10 Except for the parameters `packet-buffer-available` and `directive-invocation-online`, status parameters defined in this Recommended Practice shall not be modified as result of FSP-UNBIND, FSP-PEER-ABORT, or protocol abort.

NOTE – The parameter `directive-invocation-online` shall be set to 'no' as result of an UNBIND operation or of an abort only when the directive invocation is enabled for the service instance. If directive invocation is not enabled, the parameter shall not be modified. It can, however, change in periods when the service instance is not bound.

3.1.4.1.11 The interface `IFSP_SIUpdate` shall provide methods to retrieve the values of all status parameters. The values returned by these methods shall be undefined before configuration has been completed.

3.1.4.2 Update of Status Information by the Application

3.1.4.2.1 The service element shall provide the interface `IFSP_SIUpdate` for a provider-side service instance. This interface shall be used by the application to inform the service element of specific events in the production process.

3.1.4.2.2 When methods of this interface are called, the service element shall:

a) update the status parameters according to the arguments passed with the method invocations;

b) generate and transmit the following notifications if requested by the application and if the state of the service instance is 'ready' or 'active':

   1) `'packet processing started'`;

   2) `'packet radiated'`;

   3) `'packet acknowledged'`;

   4) `'sldu expired'`;

   5) `'packet transmission mode mismatch'`;

   6) `'transmission mode capability change'`;

   7) `'buffer empty'`;

   8) `'no invoke directive capability on this VC'`;

9) `invoke directive capability on the VC established`;

10) `positive confirm response to directive`;

11) `negative confirm response to directive`;

12) `VC aborted`;

13) `production interrupted`;

14) `production halted`;

15) `production operational`;

16) `action list completed`;

17) `action list not completed`; and

18) `event condition evaluated to false`.

NOTE – The application may opt to generate and transmit the notifications itself using the interface `ISLE_ServiceInitiate` as for other PDUs.

3.1.4.2.3 The application must inform the service element of the events defined in 3.1.4.2.1 to 3.1.4.12.1 via the interface `IFSP_SIUpdate` during the complete lifetime of the service instance, independent of the state of the service instance.

NOTE – This applies regardless of whether the application opts or not opts to generate and transmit the notifications itself using the interface `ISLE_ServiceInitiate` as for other PDUs.

3.1.4.2.4 The application should invoke the methods of the interface `IFSP_SIUpdate` when one of the events defined in 3.1.4.13.1 and 3.1.4.14.1 occurs to generate the appropriate notification and send it to the service user.

NOTES

1 The methods described in 3.1.4.2.1 to 3.1.4.12.1 update status parameters maintained by the service instance. Status information must be updated in periods in which the service user is not connected such that it is up to date following a successful BIND operation. Failure to report one of the events defined in 3.1.4.2.1 to 3.1.4.12.1 can result in inconsistent status information.

2 Generation and transmission of notifications can be disabled by a method argument if this feature is not wanted.

3 The methods described in 3.1.4.13.1 and 3.1.4.14.1 do not affect status information maintained by the service instance. Therefore, an application generating and transmitting notifications itself does not need to call these methods.

3.1.4.3 Packet Processing Started

3.1.4.3.1 The application calls the method `PacketStarted()` of the interface `IFSP_SIUpdate` whenever processing of a packet started.

3.1.4.3.2 When calling the method `PacketStarted()` the application shall provide the following information using the method arguments:

a) the identification and transmission mode of the packet for which processing started;

b) the time at which processing started; and

c) the available buffer size.

3.1.4.3.3 The method `PacketStarted()` shall:

a) increment the parameter `number-of-AD-packets-processed` or `number-of-BD-packets-processed` depending on the transmission mode of the packet;

b) update the parameters `packet-identification-last-processed` and `production-start-time` according to the arguments passed to the method;

c) set the value of the parameter `packet-status` to ‘production started’;

d) update the parameter `packet-buffer-available` according to the argument passed to the method;

e) if requested by the application, send the notification ‘packet processing started’ if the state of the service instance is ‘ready’ or ‘active’.

NOTES

1 Transmission of the notification must not be requested unless a packet processing started report has been requested for the packet by the service user. This cannot be checked by the service element.

2 Because of performance considerations, the method does not perform any checks. The application must ensure that the preconditions specified in A4.3 are fulfilled.

3.1.4.4 Packet Radiated

3.1.4.4.1 The application shall call the method `PacketRadiated()` of the interface `IFSP_SIUpdate` whenever a packet completed radiation.

3.1.4.4.2 When calling the method `PacketRadiated()` the application shall provide the following information using the method arguments:

a) the identification and transmission mode of the packet for which radiation completed;

b) the time at which radiation completed.

3.1.4.4.3 The method `PacketRadiated()` shall:

a) increment the parameter `number-of-AD-packets-radiated` or `number-of-BD-packets-radiated` depending on the transmission mode of the packet;

b) if the transmission mode of the packet is 'expedited' set the parameters `packet-identification-last-OK` and `production-stop-time` according to the arguments passed to the method;

c) if the identification of the radiated packet equals the parameter `packet-identification-last-processed`, set the parameter `packet-status` to 'radiated';

d) if requested by the application, send the notification 'packet radiated' if the state of the service instance is 'ready' or 'active'.

NOTES

1 Transmission of the notification must not be requested unless a packet radiated report has been requested for the packet by the service user. This cannot be checked by the service element.

2 Because of performance considerations, the method shall not perform any checks. The application must ensure that the preconditions specified in A4.3 are fulfilled.

3.1.4.5 Packet Acknowledged

3.1.4.5.1 The application shall call the method `PacketAcknowledged()` whenever all components of a packet have been acknowledged by the space element via the associated stream of CLCW.

3.1.4.5.2 When calling the method `PacketAcknowledged()` the application shall provide the following information using the method arguments:

a) the identification of the packet that was acknowledged;

b) the time at which the packet was acknowledged.

3.1.4.5.3 The method `PacketAcknowledged()` shall:

a) increment the parameter `number-of-packets-acknowledged`;

b) set the parameters `packet-identification-last-OK` and `production-stop-time` according to the arguments passed to the method;

c) if the identification of the acknowledged packet equals the parameter `packet-identification-last-processed`, set the parameter `packet-status` to 'acknowledged';

d) if requested by the application, send the notification 'packet acknowledged' if the state of the service instance is 'ready' or 'active'.

NOTES

1 Transmission of the notification must not be requested unless a packet acknowledged report has been requested for the packet by the service user. This cannot be checked by the service element.

2 Because of performance considerations, the method shall not perform any checks. The application must ensure that the preconditions specified in A4.3 are fulfilled.

3.1.4.6 Packet Buffer Empty

3.1.4.6.1 The application shall call the method `BufferEmpty()` whenever the packet buffer becomes empty because all packets were processed.

3.1.4.6.2 The method `BufferEmpty()` shall:

a) set the parameter `packet-buffer-available` to the value of the parameter `maximum-packet-buffer`, defined in 3.1.2.4;

b) if requested by the application, send the notification 'buffer empty' if the state of the service instance is 'ready' or 'active'.

NOTE – The method must not be called when the packet buffer is cleared because of one of the events for which reference [3] requires discarding of buffered packets.

3.1.4.7 Failure to Start Packet Processing

3.1.4.7.1 The application shall call the method `PacketNotStarted()` whenever processing of a packet could not be started, because:

a) the latest production start time was expired;

b) the production status was 'interrupted'; or

c) the required transition mode capability was not available.

3.1.4.7.2 When calling the method `PacketNotStarted()` the application shall provide the following information using the method arguments:

a) the identification and transmission mode of the packet which could not be started;

b) the time at which processing was attempted;

c) the reason why processing could not be started;

d) a list of identifiers of all packets that were or will be discarded because of the problem identified, excluding the packet for which the failure is reported;

NOTE – The service element shall always insert the packet that could not be started into the FSP-ASYNC-NOTIFY parameter `packet-identification-list`. Therefore, this packet must not be included into the list provided by the caller. If the packet for which failure is reported is the only packet that is discarded the list shall not be supplied.

e) the available buffer size.

3.1.4.7.3 The method `PacketNotStarted()` shall:

a) increment the parameter `number-of-AD-packets-processed` or `number-of-BD-packets-processed` depending on the transmission mode of the packet;

b) set the parameters `packet-identification-last-processed` and `production-start-time` according to the arguments passed to the method;

c) set the parameter `packet-status` according to reason supplied by the application:

   1) if the packet could not be started because the latest production start time expired, the `packet-status` shall be set to 'expired';

   2) if the packet could not be started because the production status was 'interrupted', the `packet-status` shall be set to 'production not started';

   3) if the packet could not be started because the required transmission mode capability was not available, the `packet-status` shall be set to 'unsupported transmission mode';

d) update the parameter `packet-buffer-available` according to the argument passed to the method;

e) if requested by the application, and if the state of the service instance is 'ready' or 'active':

   1) if the packet could not be started because the latest production start time expired, the notification 'sldu expired' shall be sent;

   2) if the packet could not be started because the production status was 'interrupted', the notification 'production interrupted' shall be sent;

   3) if the packet could not be started because the required transmission mode capability was not available, the notification 'transmission mode mismatch' shall be sent.

3.1.4.8 Production Status Change

3.1.4.8.1 The application shall call the method `ProductionStatusChange()` whenever the production status changes, including changes of the operational sub-states.

3.1.4.8.2 If the change of the production status was caused by execution of the directive 'abort VC', the method `VCAborted()` shall be called instead of the method `ProductionStatusChange()`.

3.1.4.8.3 When calling the method `ProductionStatusChange()` the application shall provide the following information using the method arguments:

a) the new value of the production status;

b) the FOP alert if the sequence-controlled service is terminated;

c) a list of identifiers of all packets that were or will be discarded because of the production status change;

NOTE – If no packets need to be discarded because of production status change, the list shall not be supplied.

d) the available buffer size.

3.1.4.8.4 The method `ProductionStatusChange()` shall:

a) set the parameter `production-status` to the value supplied by the argument;

b) if the value of the parameter `packet-identification-last-processed` is contained in the list of discarded packets supplied by the application:

   1) if the new production status is 'interrupted' or 'halted', set the parameter `packet-status` to 'interrupted';

   2) if the new production status is 'operational BD' or 'operational AD suspended', set the parameter `packet-status` to 'unsupported transmission mode';

NOTE – In all other cases, no packets shall be affected and the list of discarded packets shall not be present.

c) update the parameter packet-buffer-available according to the argument passed to the method;

d) if requested by the application, and if the state of the service instance is 'ready' or 'active':

   1) if the production status changed to 'halted', send the notification 'production halted';

2) if the production status changed from ‘configured’ or ‘operational’ to ‘interrupted’ and the list of discarded packets is present and not empty, send the notification ‘production interrupted’;

3) if the production status changed from ‘operational AD and BD’ to ‘interrupted’ and the list of discarded packets is not present or empty, send the notification ‘transmission mode capability change’;

4) if the production status changed from ‘configured’ or ‘interrupted’ to ‘operational’, and the production status last reported to the user is not ‘operational’, send the notification ‘production operational’;

5) if the production status changed from ‘operational AD and BD’ to ‘operational BD’ or ‘operational AD suspended’ or if the production status changed from ‘operational BD’ or ‘operational AD suspended’ to ‘operational AD and BD’, send the notification ‘transmission mode capability change’;

6) if the production status changed from ‘operational AD and BD’ to ‘operational BD’ or ‘operational AD suspended’ and the list of discarded packets is present and not empty, send the notification ‘packet transmission mode mismatch’.

NOTE – If this condition is true, the notification ‘packet transmission mode mismatch’ shall be sent in addition to and after the notification ‘transmission mode capability change’.

3.1.4.8.5 The method `ProductionStatusChange()` shall perform the following consistency checks. If any of the checks fail, the method shall return an error code and perform no actions:

a) when the production status changes from ‘configured’ or ‘interrupted’ to ‘operational’, the sub-state must be ‘BD’;

b) when the production status changes from operational to ‘interrupted’ or ‘halted’ and the status of the packet last processed is ‘processing started’, the list of affected packets must be supplied and the packet last processed must be a member of this list;

c) when the list of affected packets is supplied and is not empty, the following conditions must hold:

1) the production status must have changed;

2) the new production status must not be ‘operational AD and BD’ or ‘configured’;

3) the old production status must have been ‘operational’;

4) the production status must not have changed from ‘operational AD suspended’ to ‘operational BD’.

3.1.4.8.6 If the checks identified in 3.1.4.8.5 succeed but the production status has not changed, the method shall perform no actions and shall inform the caller accordingly.

3.1.4.9 Reported Production Status

Whenever the service element sends one of the notifications ‘production operational’, ‘production interrupted’, or ‘production halted’, it shall memorize the reported status.

NOTE – This ‘reported production status’ shall be used to prevent that the notification ‘production operational’ is sent to a user that was not informed of a change to a non operational status either because the service instance was not bound when the change occurred or because no packets were affected by the production status ‘interrupted’.

3.1.4.10 Execution of the Directive ‘Abort VC’

3.1.4.10.1 The application shall call the method `VCAborted()` whenever the directive ‘abort VC’ was executed on the virtual channel passing a list of identifiers of all packets that were discarded.

3.1.4.10.2 When calling the method `VCAborted()` the application shall provide the following information using the method arguments:

a) a list of identifiers of all packets that were or will be discarded because of processing of the directive;

NOTE – If no packets need to be discarded because of production status change, the list shall not be supplied.

b) the available buffer size.

3.1.4.10.3 The method `VCAborted()` shall:

a) set the parameter `production-status` to ‘operational BD’;

b) if the value of the parameter `packet-identification-last-processed` is contained in the list of discarded packets supplied by the application and the transmission mode of that packet is ‘sequence controlled’, set the parameter packet status to ‘interrupted’;

c) update the parameter `packet-buffer-available` according to the argument passed to the method;

d) if requested by the application, send the notification ‘VC aborted’ if the state of the service instance is ‘ready’ or ‘active’.

NOTE – If no packets were buffered or were being processed when the directive was executed, the list of discarded packets shall not be supplied.

3.1.4.11 No Invoke Directive Capability

3.1.4.11.1 The application shall call the method `NoDirectiveCapability()` when the service instance for which directive invocation is enabled is no longer bound because of an UNBIND operation, a PEER-ABORT operation or a protocol abort event.

3.1.4.11.2 If directive invocation is not enabled for the service instance the method `NoDirectiveCapability()` shall:

a) set the parameter `directive-invocation-online` to ‘no’;

b) if requested by the application and if the state of the service instance is ‘ready’ or ‘active’ transmit the notification ‘no invoke directive capability on this VC’.

3.1.4.11.3 If directive invocation is enabled for the service instance, the method shall perform no actions and shall inform the caller accordingly.

3.1.4.12 Invoke Directive Capability Established

3.1.4.12.1 The application shall call the method `DirectiveCapabilityOnline()` when a service instance for which directive invocation is enabled has bound to the provider.

3.1.4.12.2 If directive invocation is not enabled for the service instance, the method `DirectiveCapabilityOnline()` shall:

a) set the parameter `directive-invocation-online` to ‘yes’;

b) if requested by the application and if the state of the service instance is ‘ready’ or ‘active’ transmit the notification ‘invoke directive capability on this VC established’.

3.1.4.12.3 If directive invocation is enabled for the service instance, the method shall perform no actions and shall inform the caller accordingly.

3.1.4.13 Directive Execution Completed

3.1.4.13.1 The application should call the method `DirectiveCompleted()` when execution of a directive invoked by the operation FSP-INVOKE-DIRECTIVE completes.

3.1.4.13.2 When calling the method `DirectiveCompleted()` the application shall provide the following information using the method arguments:

a) the directive identification copied from the FSP-INVOKE-DIRECTIVE invocation;

b) the result of execution, indicating whether execution succeeded (‘positive result’) or failed (‘negative result’);

c) in case of a negative result, the FOP alert providing the reason for the failure.

3.1.4.13.3 If requested by the caller, the method `DirectiveCompleted()` shall:

a) send the notification 'positive confirm result to directive' if the result is positive;

b) send the notification 'negative confirm result to directive' if the result is negative.

3.1.4.14 Event Processing Completed

3.1.4.14.1 The application should call the method `EventProcCompleted()` when processing of an event requested by an accepted FSP-THROW-EVENT operation completes.

3.1.4.14.2 When calling the method `EventProcCompleted()` the application shall provide the following information using the method arguments:

a) the event invocation identification as copied from the FSP-THROW-EVENT invocation;

b) the result of execution, indicating whether:

   1) the action list associated with the event was completely executed;

   2) at least one of the actions in the associated action list failed; or

   3) the condition associated with the event evaluated to false.

3.1.4.14.3 If requested by the caller, the method `EventProcCompleted()` shall:

a) send the notification 'action list completed' if the action list associated with the event was completely executed;

b) send the notification 'action list not completed' if at least one of the actions in the associated action list failed;

c) send the notification 'event action evaluated to false' if the condition associated with the event evaluated to false.

3.1.4.15 Consistency Checks

The service element shall apply the following rules for checking of consistency:

a) The methods `PacketStarted()`, `PacketRadiated()`, `PacketAcknowledged()`, and `BufferEmpty()` shall perform no checks.

NOTE – These methods must be called frequently during nominal operation. Because of performance considerations, the service element shall fully rely on the application to ensure that the methods are used correctly. Detailed preconditions are defined in A4.3.

CCSDS HISTORICAL DOCUMENT

b) The methods `DirectiveCompleted()` and `EventProcCompleted()` shall perform no checks.

NOTE – Checking of correctness of these method invocations requires information not available to the service element. Therefore, the service element must fully rely on the application to ensure that the methods are used correctly.

c) For other methods, the service element shall verify that the arguments are consistent and that the method call is consistent with the values of the status parameters before the method was invoked. If the check fails, the service element shall proceed as follows:

   1) if applying the update results in a consistent set of status parameters, the service element shall perform the update and shall send the notification (if requested) but shall return an error code to the application as a warning;

   2) if an update would result in inconsistent status parameters, the service element shall not perform the update, shall not send any notifications, but shall return an appropriate error code.

NOTE – Further details concerning the checks performed and return codes passed to the caller are defined in A4.3.

3.1.5 PROCESSING OF FSP PROTOCOL DATA UNITS

NOTES

1 The service element processes FSP PDUs according to the general specifications in reference [5]. This subsection only addresses additional checks and processing steps defined for the FSP service. FSP-specific checks defined in reference [3] but not listed in this subsection must be performed by the application. Subsection 2.2.9 provides a discussion of the borderline between the application and the service element.

2 3.1.4 defines processing requirements for update of status information and generation of notifications. Annex subsection A3 defines the checks that operation objects perform when the methods `VerifyInvocationArguments()` and `VerifyReturnArguments()` are called. Reference [5] contains specifications defining how the service element handles error codes returned by these methods.

3.1.5.1 FSP TRANSFER DATA

3.1.5.1.1 When receiving an FSP–TRANSFER–DATA invocation, the service element shall perform the following checks in addition to the checks defined in reference [5] for all PDUs. These checks shall be performed in the specified sequence:

CCSDS HISTORICAL DOCUMENT

CCSDS HISTORICAL DOCUMENT

a) If the ‘earliest production time’ and the ‘latest production time’ are both specified, the ‘earliest production time’ must not be later than the ‘latest production time’.

b) The time window defined by the ‘earliest production time’ and the ‘latest production time’ must overlap with the provision period of the service instance.

c) If the configuration parameter `segment-header` defines that segment headers are used, the value of the parameter ‘MAP identifier’ must be contained in the configured ‘map list’. If segment headers are not used, the value of the ‘MAP identifier’ must be ‘none’.

d) The value of the parameter `transmission-mode` must match the configured permitted transmission mode.

e) If the `transmission-mode` is ‘expedited’, the ‘acknowledged notification’ must not be requested.

f) The size of the packet contained in the PDU must not be larger than the value of the configuration parameter `maximum-packet-length` allows.

3.1.5.1.2 If any of the checks defined in 3.1.5.1.1 fail, or a return PDU with a negative result must be generated because a check defined in reference [5] failed, the service element shall proceed as follows:

a) if the service element can guarantee that all preceding FSP–TRANSFER–DATA invocations have already been processed by the application, or that the PDU processed by the service element is the first FSP–TRANSFER–DATA invocation following START, the service element may generate a FSP–TRANSFER–DATA return with a negative result and transmit that to the service user;

NOTE – In that case, the service element shall use the status parameters ‘expected packet identification’ and `packet-buffer-available` to set the parameters of the FSP–TRANSFER–DATA return.

b) if the conditions defined in 3.1.5.1.2 item a) are not met or cannot be verified, the service element shall set the result parameter to ‘negative’, set the appropriate diagnostic in the operation object, and pass the operation object to the application;

c) in order to ensure that the result parameter of the operation object always has a valid reading, the service element shall set the result parameter to ‘positive’ if all checks performed by the service element succeeded.

NOTES

1 It is noted that this processing deviates from the standard way in which confirmed PDUs are handled by the service element. The reasons for this specification are explained in 2.2.9.3.

CCSDS HISTORICAL DOCUMENT

2 An implementation is not required to generate and transmit a FSP–TRANSFER–DATA return also when it could verify that the conditions defined in 3.1.5.1.2 item a) are met. A service element can use one of the following approaches:

– ensure that no FSP–TRANSFER–DATA invocations are queued between the service element and the application, and never pass an invocation for which a check has failed to the application;

– always pass FSP–TRANSFER–DATA invocations to the application; or

– decide on a case-by-case basis.

3 Implementations should document the approach used. Applications should always expect the service element to pass FSP–TRANSFER–DATA invocations with a negative result if substitutability of SLE API components shall be maintained.

4 Processing expected from the application is defined in 3.3.

3.1.5.2 FSP THROW EVENT

3.1.5.2.1 If an FSP–THROW–EVENT return PDU with a negative result must be generated because a check defined in reference [5] failed, the service element shall proceed as defined by the following specifications.

3.1.5.2.2 If the service element can guarantee that all preceding FSP–THROW–EVENT invocations have already been processed by the application or that the PDU processed by the service element is the first FSP–THROW–EVENT invocation following BIND, the service element may generate a FSP–THROW–EVENT return with a negative result and transmit that to the service user.

NOTE – In that case, the service element shall use the status ‘expected event invocation identifier’ to set the parameter of the FSP–THROW–EVENT return.

3.1.5.2.3 If the conditions defined in 3.1.5.2.2 are not met or cannot be verified the service element shall set the result parameter to ‘negative’, set the appropriate diagnostic in the operation object, and pass the operation object to the application.

3.1.5.2.4 In order to ensure that the result parameter of the operation object always has a valid reading, the service element shall set the result parameter to ‘positive’ if all checks performed by the service element succeeded.

NOTES

1 It is noted that this processing deviates from the standard way in which confirmed PDUs are handled by the service element. The reasons for this specification are explained in 2.2.9.3.

2 A service element shall not be required to generate and transmit a FSP–THROW–EVENT return also when it could verify that the conditions defined in 3.1.5.2.2 are met. A service element can use one of the following approaches:

– ensure that no FSP–THROW–EVENT invocations are queued between the service element and the application, and never pass an invocation for which a check has failed to the application;

– always pass FSP–THROW–EVENT invocations to the application; or

– decide on a case-by-case basis.

3 Implementations should document the approach used. Applications should always expect that the service element pass FSP–THROW–EVENT invocations with a negative result if substitutability of SLE API components shall be maintained.

4 Processing expected from the application is defined in 3.3.

3.1.5.3 FSP INVOKE DIRECTIVE

3.1.5.3.1 When receiving an FSP–INVOKE-DIRECTIVE invocation, the service element shall verify that invocation of directives is enabled for the service instance.

3.1.5.3.2 If invocation of directives is not enabled for the service instance, the service element shall not pass the invocation to the application but shall send a return with a negative result and the appropriate diagnostic.

3.1.5.3.3 If the invocation of directives is enabled for the service instance, but a FSP-INVOKE-DIRECTIVE return PDU with a negative result must be generated because a check defined in reference [5] failed, the service element shall proceed as follows:

a) If the service element can guarantee that all preceding FSP–INVOKE-DIRECTIVE invocations have already been processed by the application, or that the PDU processed by the service element is the first FSP–INVOKE-DIRECTIVE invocation following BIND, the service element may generate a FSP–INVOKE-DIRECTIVE return with a negative result and transmit that to the service user.

b) In that case, the service element uses the status parameter expected directive invocation identifier to set the parameter of the FSP–INVOKE-DIRECTIVE return.

c) If the conditions defined in 3.1.5.3.3 a) are not met or cannot be verified, the service element shall set the result parameter to ‘negative’, set the appropriate diagnostic in the operation object, and pass the operation object to the application.

d) In order to ensure that the result parameter of the operation object always has a valid reading, the service element shall set the result parameter to ‘positive’ if all checks performed by the service element succeeded.

NOTES

1 It is noted that this processing deviates from the standard way in which confirmed PDUs are handled by the service element. The reasons for this specification are explained in 2.2.9.3.

2 An implementation is not required to generate and transmit a FSP–INVOKE-DIRECTIVE return also when it could verify that the conditions defined in 3.1.5.3.3 item a) are met. A service element can use one of the following approaches:

– ensure that no FSP–INVOKE-DIRECTIVE invocations are queued between the service element and the application, and never pass an invocation for which a check has failed to the application;

– always pass FSP–INVOKE-DIRECTIVE invocations to the application; or

– decide on a case-by-case basis.

3 Implementations should document the approach used. Applications should always expect the service element to pass FSP–INVOKE-DIRECTIVE invocations with a negative result if substitutability of SLE API components shall be maintained.

4 Processing expected from the application is defined in 3.3.

3.1.6 SERVICE INSTANCE SPECIFIC OPERATION FACTORY

For FSP service instances, the interface `ISLE_SIOpFactory` specified in reference [5] shall support creation and configuration of operation objects for all operations specified in 3.2 with exception of the object for the operation IFSP–STATUS–REPORT.

NOTE – The initial values of parameters that shall be set for FSP-specific operation objects are defined in annex A. The operation IFSP–STATUS–REPORT shall be handled autonomously by the provider-side service element. There is no need for the application to create this object.

3.2 SLE OPERATIONS

3.2.1 The component ‘SLE Operations’ shall provide operation objects for the following FSP operations in addition to those specified in reference [5]:

a) FSP–START;

b) FSP–TRANSFER–DATA;

c) FSP–ASYNC–NOTIFY;

CCSDS HISTORICAL DOCUMENT

d) FSP–STATUS–REPORT;

e) FSP–GET–PARAMETER;

f) FSP–THROW–EVENT;

g) FSP-INVOKE-DIRECTIVE.

3.2.2 The operation factory shall create the operation objects specified in 3.2.1 when the requested service type is FSP.

3.2.3 The operation factory shall additionally create the following operation objects specified in reference [5] when the requested service type is FSP:

a) SLE–BIND;

b) SLE–UNBIND;

c) SLE–PEER–ABORT;

d) SLE–STOP;

e) SLE–SCHEDULE–STATUS–REPORT.

3.3 SLE APPLICATION

NOTE – This subsection summarizes specific obligations of a FSP provider application using the SLE API.

3.3.1 CONFIGURATION AND UPDATE OF STATUS INFORMATION

3.3.1.1 Following creation of a service instance, and setting of the configuration parameters defined in reference [5], the application shall set the configuration parameters defined in 3.1.2 via the interface `IFSP_SIAdmin`.

3.3.1.2 Following creation of a service instance, the application shall initialize the FOP parameters defined in 3.1.3 via the interface `IFSP_FOPMonitor` and subsequently update these parameters whenever a change occurs.

3.3.1.3 The application shall inform the service element of all events defined in 3.1.4.2 by invocation of the appropriate methods of the interface `IFSP_SIUpdate`.

CCSDS HISTORICAL DOCUMENT

3.3.2 PROCESSING OF FSP TRANSFER DATA

When receiving a FSP–TRANSFER–DATA invocation via the interface ISLE_ServiceInform, the application shall check the result parameter of the operation object and shall perform the following steps:

a) if the result is negative, the application shall set the expected packet identification and the available buffer size and then passes the operation back to the service element using the method InitiateOpReturn() of the interface ISLE_ServiceInitiate;

b) if the result is positive, the application shall perform the checks not specified in 3.1.5 and reference [5].

   1) If any of these checks fail, the application shall set the appropriate diagnostic, the expected packet identification, and the available buffer size and then pass the operation object to the service element using the method InitiateOpReturn() in the interface ISLE_ServiceInitiate.

   2) If all checks succeed, the application stores the packet to the packet buffer, the application shall set a positive result, the expected packet identification, and the available buffer size and then pass the operation object back to the service element using the method InitiateOpReturn() in the interface ISLE_ServiceInitiate.

3.3.3 PROCESSING OF FSP THROW EVENT

When receiving a FSP–THROW–EVENT invocation via the interface ISLE_ServiceInform, the application shall check the result parameter of the operation object and perform the following steps:

a) if the result is negative, the application shall set the expected event invocation and pass the operation back to the service element using the method InitiateOpReturn() in the interface ISLE_ServiceInitiate;

b) if the result is positive, the application performs the checks not specified in 3.1.5 and reference [5]:

   1) if any of these checks fail, the application shall set the appropriate diagnostic and the expected event invocation identifier and then pass the operation object to the service element using the method InitiateOpReturn() in the interface ISLE_ServiceInitiate;

   2) if all checks succeed, the application shall perform the required operation, set a positive result, and the expected event invocation identifier and then pass the operation object back to the service element using the method InitiateOpReturn() in the interface ISLE_ServiceInitiate.

3.3.4 PROCESSING OF FSP INVOKE DIRECTIVE

When receiving a FSP–INVOKE-DIRECTIVE invocation via the interface `ISLE_ServiceInform`, the application shall check the result parameter of the operation object and perform the following steps:

a) if the result is negative, the application shall set the expected directive invocation identifier and pass the operation back to the service element using the method `InitiateOpReturn()` in the interface `ISLE_ServiceInitiate`;

b) if the result is positive, the application shall perform the checks required:

   1) if any of these checks fail, the application shall set the appropriate diagnostic and the expected directive invocation identifier and then pass the operation object to the service element using the method `InitiateOpReturn()` in the interface `ISLE_ServiceInitiate`;

   2) if all checks succeed, the application shall perform the required operation, set a positive result, and the expected directive invocation identifier, and then pass the operation object back to the service element using the method `InitiateOpReturn()` in the interface `ISLE_ServiceInitiate`.

3.4 SEQUENCE OF DIAGNOSTIC CODES

3.4.1 GENERAL

3.4.1.1 Reference [3] requires provider applications that do not perform checks in the sequence of the diagnostic codes defined in the specification to document the sequence in which checks are actually performed.

3.4.1.2 The specification in 3.1.5 does not preserve the sequence of the diagnostic codes defined in reference [3] for the operation FSP–TRANSFER–DATA. This subsection defines the actual sequence of checks performed by the API Service Element. For the checks that remain to be performed by the provider application, the sequence defined in reference [3] is maintained. Applications applying a different sequence must provide a modified documentation.

3.4.2 SEQUENCE OF FSP-TRANSFER-DATA DIAGNOSTIC CODES

3.4.2.1 Codes Set by the API Service Element

a) ‘duplicate invoke id’;

b) ‘inconsistent time range’;

c) ‘invalid time’;

d) ‘invalid MAP’;

e) ‘invalid notification request’; and

f) ‘packet too long’.

3.4.2.2 Codes Set by the Application

a) ‘unable to process’;

b) ‘unable to store’;

c) ‘out of sequence’;

d) ‘duplicate packet identification’;

e) ‘invalid time’;

f) ‘conflicting production time intervals’;

g) ‘late sldu’;

h) ‘invalid delay time’;

i) ‘invalid transmission mode’;

j) ‘unsupported packet version’;

k) ‘incorrect packet type’;

l) ‘invalid packet apid’; and

m) ‘other reason’.

CCSDS HISTORICAL DOCUMENT

ANNEX A

FSP SPECIFIC INTERFACES

(Normative)

A1 INTRODUCTION

This annex specifies FSP specific

a) data types;

b) interfaces for operation objects; and

c) interfaces for service instances.

The specification of the interfaces follows the design patterns, conventions and the additional conventions described in reference [4].

The presentation uses the notation and syntax of the C++ programming language as specified in reference [5].

CCSDS HISTORICAL DOCUMENT

A2 FSP TYPE DEFINITIONS

File `FSP_Types.h`

The following types have been derived from the ASN.1 module CCSDS-SLE-TRANSFER-FSP-STRUCTURES in reference [3]. The source ASN.1 type is indicated in brackets. For all enumeration types a special value 'invalid' is defined, which is returned if the associated value in the operation object has not yet been set, or is not applicable in case of a choice.

Absolute Priority [AbsolutePriority]

```
typedef struct FSP_AbsolutePriority
{
  unsigned int mapOrVc;         /* 0 to 63 */
  unsigned int priority;        /* 1 (highest) to 64 (lowest) */
} FSP_AbsolutePriority;
```

An entry in the priority list used for multiplexing of MAPs and VCs.

Blocking Usage [BlockingUsage]

```
typedef enum FSP_BlockingUsage
{
  fspAU_permitted         =  0,
  fspAU_notPermitted      =  1,
  fspAU_invalid           = -1
} FSP_BlockingUsage;
```

Application Identifier (APID)

```
typedef unsigned long FSP_ApId;       /* 0 to 2047 */
```

Packet Buffer Size [BufferSize]

```
typedef unsigned long FSP_BufferSize;
```

Size of the packet buffer or the remaining free space in the buffer measured in octets.

Packet Identification [PacketIdentification]

```
typedef unsigned long FSP_PacketId;
```

Directive Identification

```
typedef unsigned long FSP_DirectiveId;
```

Identifier of a Thrown Event [EventInvocationId]

```
typedef unsigned long FSP_EventInvocationId;
```

Directive [FspInvokeDirectiveInvocation]

```
typedef enum FSP_Directive
{
  fspDV_initiateADwithoutCLCW        =  0,
  fspDV_initiateADwithCLCW           =  1,
  fspDV_initiateADwithUnlock         =  2,
  fspDV_initiateADwithSetVR          =  3,
  fspDV_terminateAD                  =  4,
  fspDV_resumeAD                     =  5,
  fspDV_setVS                        =  6,
  fspDV_setFopSlidingWindow          =  7,
  fspDV_setT1Initial                 =  8,
  fspDV_setTransmissionLimit         =  9,
  fspDV_setTimeoutType               = 10,
  fspDV_abortVC                      = 11,
  fspDV_modifyMapMuxControl          = 12,
  fspDV_invalid                      = -1
} FSP_Directive;
```

MAP or VC Identification [MapOrVcId]

```
typedef unsigned int FSP_MapOrVcId;    /* 0 to 63 */
```

MAP Identification [MapId]

```
typedef FSP_MapOrVcId FSP_MapId;
```

VC Identification

```
typedef FSP_MapOrVcId FSP_VcId;
```

Multiplexing Scheme [MuxScheme]

```
typedef enum FSP_MuxScheme
{
  fspMS_fifo               =  0,
  fspMS_absolutePriority   =  1,
  fspMS_pollingVector      =  2,
  fspMS_invalid            = -1
} FSP_MuxScheme;
```

Timeout Type [FspGetParameter]

```
typedef enum FSP_TimeoutType
{
  fspTT_generateAlert      =  0,
  fspTT_suspendAD          =  1,
  fspTT_invalid            = -1
} FSP_TimeoutType;
```

Timeout Type [FspInvokeDirectiveParameter]

```
typedef enum FSP_DirectiveTimeoutType
{
  fspDTT_terminateAD       =  0,
  fspDTT_suspendAD         =  1,
  fspDTT_invalid           = -1
} FSP_DirectiveTimeoutType;
```

The Timeout Type used in the Invoke Directive invocation.

Transmission Mode [TransmissionMode]

```
typedef enum FSP_TransmissionMode
{
  fspTM_sequenceControlled        =  0,   /* AD mode    */
  fspTM_expedited                 =  1,   /* BD mode    */
  fspTM_sequenceControlledUnblock =  2,   /* unblock AD */
  fspTM_invalid                   = -1
} FSP_TransmissionMode;
```

Transmission Mode [PermittedTransmissionMode]

```
typedef enum FSP_PermittedTransmissionMode
{
  fspPTM_sequenceControlled        =  0,
  fspPTM_expedited                 =  1,
  fspPTM_any                       =  2,
  fspPTM_invalid                   = -1
} FSP_PermittedTransmissionMode;
```

FSP Start Diagnostic [DiagnosticFspStart]

```
typedef enum FSP_StartDiagnostic
{
  fspSTD_outOfService           =   0,
  fspSTD_unableToComply         =   1,
  fspSTD_productionTimeExpired =   2,
  fspSTD_invalid                =  -1
} FSP_StartDiagnostic;
```

FSP Transfer Data Diagnostic [DiagnosticFspTransferData]

```
typedef enum FSP_TransferDataDiagnostic
{
  fspXFD_unableToProcess                    =   0,
  fspXFD_unableToStore                      =   1,
  fspXFD_packetIdOutOfSequence              =   2,
  fspXFD_duplicatePacketIdentification      =   3,
  fspXFD_inconsistentTimeRange              =   4,
  fspXFD_invalidTime                        =   5,
  fspXFD_conflictingProductionTimeIntervals =   6,
  fspXFD_lateSldu                           =   7,
  fspXFD_invalidDelayTime                   =   8,
  fspXFD_invalidTransmissionMode            =   9,
  fspXFD_invalidMap                         =  10,
  fspXFD_invalidNotificationRequest         =  11,
  fspXFD_packetTooLong                      =  12,
  fspXFD_unsupportedPacketVersion           =  13,
  fspXFD_incorrectPacketType                =  14,
  fspXFD_invalidPacketApid                  =  15,
  fspXFD_invalid                             = -1
} FSP_TransferDataDiagnostic;
```

FSP Get Parameter Diagnostic [DiagnosticFspGetParameter]

```
typedef enum FSP_GetParameterDiagnostic
{
  fspGP_unknownParameter =   0,
  fspGP_invalid         =   -1
} FSP_GetParameterDiagnostic;
```

FSP Invoke Directive Diagnostic [DiagnosticFspInvokeDirective]

```
typedef enum FSP_InvokeDirectiveDiagnostic
{
  fspID_directiveInvocationNotAllowed        =   0,
  fspID_directiveIdentificationOutOfSequence =   1,
  fspID_directiveError                       =   2,
  fspID_invalid                              =  -1
} FSP_InvokeDirectiveDiagnostic;
```

FSP Throw Event Diagnostic [DiagnosticFspThrowEvent]

```
typedef enum FSP_ThrowEventDiagnostic
{
  fspTED_operationNotSupported      =   0,
  fspTED_outOfSequence              =   1,
  fspTED_noSuchEvent                =   2,
  fspTED_invalid                    =  -1
} FSP_ThrowEventDiagnostic;
```

FSP Service Parameters [FspParameterName]

```
typedef enum FSP_ParameterName
{
  fspPN_blockingTimeoutPeriod          =   0,
  fspPN_blockingUsage                  =   1,
  fspPN_apidList                       =   2,
  fspPN_deliveryMode                   =   6,
  fspPN_directiveInvocationEnabled     =   7,
  fspPN_expectedDirectiveId            =   8,
  fspPN_expectedEventInvocationId      =   9,
  fspPN_expectedSlduIdentification     =  10,
  fspPN_fopSlidingWindow               =  11,
  fspPN_fopState                       =  12,
  fspPN_mapList                        =  16,
  fspPN_mapMuxControl                  =  17,
  fspPN_mapMuxScheme                   =  18,
  fspPN_maximumFrameLength             =  19,
  fspPN_maximumPacketLength            =  20,
  fspPN_reportingCycle                 =  26,
  fspPN_returnTimeoutPeriod            =  29,
  fspPN_segmentHeader                  =  32,
  fspPN_timeoutType                    =  35,
  fspPN_timerInitial                   =  36,
  fspPN_transmissionLimit              =  37,
  fspPN_transmitterFrameSequenceNumber =  38,
  fspPN_vcMuxControl                   =  39,
  fspPN_vcMuxScheme                    =  40,
  fspPN_virtualChannel                 =  41,
  fspPN_permittedTransmissionMode      = 107,
  fspPN_directiveInvocationOnline      = 108,
  fspPN_invalid                        =  -1
} FSP_ParameterName;
```

The parameter name values are taken from the type `ParameterName` in CCSDS-SLE-TRANSFER-SERVICE-COMMON-TYPES.

FOP Alert [FopAlert]

```
typedef enum FSP_FopAlert
{
  fspFA_noAlert                    =  0,
  fspFA_limit                      =  1,
  fspFA_lockOutDetected            =  2,
  fspFA_synch                      =  3,
  fspFA_invalidNR                  =  4,
  fspFA_Clcw                       =  5,
  fspFA_lowerLayerOutOfSync        =  6,
  fspFA_terminateAD                =  7,
  fspFA_invalid                    = -1
} FSP_FopAlert;
```

Packet Status [PacketStatus]

```
typedef enum FSP_PacketStatus
{
  fspST_acknowledged                 = sleFDS_acknowledged,
  fspST_radiated                     = sleFDS_radiated,
  fspST_productionStarted            = sleFDS_productionStarted,
  fspST_productionNotStarted         = sleFDS_productionNotStarted,
  fspST_expired                      = sleFDS_expired,
  fspST_unsupportedTransmissionMode  = sleFDS_unsupportedTransmissionMode,
  fspST_interrupted                  = sleFDS_interrupted,
  fspST_invalid                      = -1
} FSP_PacketStatus;
```

Describes the state of the last processed packet. It is defined as a subset of the type `SLE_ForwardDuStatus` specified in reference [3].

Production Status [ProductionStatus]

```
typedef enum FSP_ProductionStatus
{
  fspPS_configured              =  0,
  fspPS_operationalBd           =  1,
  fspPS_operationalAdAndBd      =  2,
  fspPS_operationalAdSuspended  =  3,
  fspPS_interrupted             =  4,
  fspPS_halted                  =  5,
  fspPS_invalid                 = -1
} FSP_ProductionStatus;
```

The status of the FSP production engine

FOP State [FspGetParameter]

```
typedef enum FSP_FopState
{
  fspFS_active                      =  0,
  fspFS_retransmitWithoutWait       =  1,
  fspFS_retransmitWithWait          =  2,
  fspFS_initialisingWithoutBCFrame  =  3,
  fspFS_initialisingWithBCFrame     =  4,
  fspFS_initial                     =  5,
  fspFS_invalid                     = -1
} FSP_FopState;
```

FSP Event Processing Result

```
typedef enum FSP_EventResult
{
  fspER_completed        =  0, /* action list completed */
  fspER_notCompleted     =  1, /* action list not completed */
  fspER_conditionFalse   =  2  /* event condition evaluated to false */
} FSP_EventResult;
```

The result of processing a thrown event.

Notification Type [FspNotification]

```
typedef enum FSP_NotificationType
{
  fspNT_packetProcessingStarted              =  0,
  fspNT_packetRadiated                       =  1,
  fspNT_packetAcknowledged                   =  2,
  fspNT_slduExpired                          =  3,
  fspNT_packetTransmissionModeMismatch       =  4,
  fspNT_transmissionModeCapabilityChange     =  5,
  fspNT_bufferEmpty                          =  6,
  fspNT_noInvokeDirectiveCapabilityOnThisVc  =  7,
  fspNT_positiveConfirmResponseToDirective   =  8,
  fspNT_negativeConfirmResponseToDirective   =  9,
  fspNT_vcAborted                            = 10,
  fspNT_productionInterrupted                = 11,
  fspNT_productionHalted                     = 12,
  fspNT_productionOperational                = 13,
  fspNT_actionListCompleted                  = 14,
  fspNT_actionListNotCompleted               = 15,
  fspNT_eventConditionEvFalse                = 16,
  fspNT_invokeDirectiveCapabilityOnThisVC    = 17,
  fspNT_invalid                              = -1
} FSP_NotificationType;
```

FSP Failure

```
typedef enum FSP_Failure
{
  fspF_expired            =  0,
  fspF_interrupted        =  1,  /* production interrupted */
  fspF_modeMismatch       =  2   /* transmission mode mismatch */
} FSP_Failure;
```

A3 FSP OPERATION OBJECTS

A3.1 FSP START OPERATION

Name `IFSP_Start`
GUID `{1D0CBEE0-E896-11d4-9F17-00104B4F22C0}`
Inheritance: `IUnknown - ISLE_Operation - ISLE_ConfirmedOperation`
File `IFSP_Start.H`

The interface provides access to the parameters of the confirmed operation FSP START.

Synopsis

```
include <FSP_Types.h>
#include <ISLE_ConfirmedOperation.H>
interface ISLE_Time;

#define IID_IFSP_Start_DEF { 0x1d0cbee0, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_Start : ISLE_ConfirmedOperation
{
  virtual FSP_PacketId
    Get_FirstPacketId() const = 0;
  virtual const ISLE_Time*
    Get_StartProductionTime() const = 0;
  virtual const ISLE_Time*
    Get_StopProductionTime() const = 0;
  virtual FSP_StartDiagnostic
    Get_StartDiagnostic() const = 0;
  virtual void
    Set_FirstPacketId( FSP_PacketId id ) = 0;
  virtual void
    Set_StartProductionTime( const ISLE_Time& startTime ) = 0;
  virtual void
    Put_StartProductionTime( ISLE_Time* pstartTime ) = 0;
  virtual void
    Set_StopProductionTime( const ISLE_Time& stopTime ) = 0;
  virtual void
    Put_StopProductionTime( ISLE_Time* pstopTime ) = 0;
  virtual void
    Set_StartDiagnostic( FSP_StartDiagnostic diag ) = 0;
};
```

Methods

`FSP_PacketId Get_FirstPacketId() const;`

Returns the first packet identification that the provider shall expect.

```
const ISLE_Time* Get_StartProductionTime() const;
```

Returns a pointer to the production start time if that parameter has been set. If the parameter has not been specified returns a NULL pointer.

```
const ISLE_Time* Get_StopProductionTime() const;
```

Returns a pointer to the production stop time if that parameter has been set. If the parameter has not been specified returns a NULL pointer.

```
FSP_StartDiagnostic Get_StartDiagnostic() const;
```

Returns the diagnostic code.

Precondition: the result is negative, and the diagnostic type is set to ‘specific’.

```
void Set_FirstPacketId( FSP_PacketId id );
```

Sets the first packet identification the provider shall accept.

```
void Set_StartProductionTime( const ISLE_Time& startTime );
```

Sets the production start time to a copy of the input argument.

```
void Put_StartProductionTime( ISLE_Time* pstartTime );
```

Stores the input argument to the parameter production start time.

```
void Set_StopProductionTime( const ISLE_Time& stopTime );
```

Sets the production stop time to a copy of the input argument.

```
void Put_StopProductionTime( ISLE_Time* pstopTime );
```

Stores the input argument to the parameter production stop time.

```
void Set_StartDiagnostic( FSP_StartDiagnostic diag );
```

Sets the result to ‘negative’, the diagnostic type to ‘specific’, and stores the value of the diagnostic code passed by the argument.

Initial Values of Operation Parameters after Creation

Parameter	Created directly	Created by Service Instance
first packet Identification	0	0
start production time	NULL (not used)	NULL (not used)
stop production time	NULL (not used)	NULL (not used)
START diagnostic	'invalid'	'invalid'

Checking of Invocation Parameters

No checks beyond those defined by inherited interfaces are performed.

Checking of Return Parameters

Parameter	Required condition
start production time	must not be NULL; if the start and the stop time are used, must be earlier than stop time
stop production time	if the start and the stop time are used, must be later than stop time
START diagnostic	must not be 'invalid' if the result is 'negative' and the diagnostic type is 'specific'

Additional Return Codes for `VerifyReturnArguments`

`SLE_E_MISSINGARG` specification that the start production time is missing

A3.2 FSP TRANSFER DATA OPERATION

Name IFSP_TransferData
GUID {91DCEBA0-E896-11d4-9F17-00104B4F22C0}
Inheritance: IUnknown – ISLE_Operation – ISLE_ConfirmedOperation
File IFSP_TransferData.H

The interface provides access to the parameters of the confirmed operation FSP-TRANSFER-DATA.

Synopsis

```
#include <FSP_Types.h>
#include <ISLE_ConfirmedOperation.H>
interface ISLE_Time;

#define IID_IFSP_TransferData_DEF { 0x91dceba0, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_TransferData : ISLE_ConfirmedOperation
{
  virtual FSP_PacketId
    Get_PacketId() const = 0;
  virtual FSP_PacketId
    Get_ExpectedPacketId() const = 0;
  virtual const ISLE_Time*
    Get_EarliestProdTime() const = 0;
  virtual const ISLE_Time*
    Get_LatestProdTime() const = 0;
  virtual SLE_Duration
    Get_DelayTime() const = 0;
  virtual FSP_TransmissionMode
    Get_TransmisionMode() const = 0;
  virtual bool
    Get_MapIdUsed() const = 0;
  virtual FSP_MapId
    Get_MapId() const = 0;
  virtual SLE_YesNo
    Get_Blocking() const = 0;
  virtual SLE_SlduStatusNotification
    Get_ProcessingStartedNotification() const = 0;
  virtual SLE_SlduStatusNotification
    Get_RadiatedNotification() const = 0;
  virtual SLE_SlduStatusNotification
    Get_AcknowledgedNotification() const = 0;
  virtual const SLE_Octet*
    Get_Data( size_t& length ) const = 0;
  virtual SLE_Octet*
    Remove_Data( size_t& length ) = 0;
  virtual FSP_BufferSize
    Get_PacketBufferAvailable() const = 0;
  virtual FSP_TransferDataDiagnostic
    Get_TransferDataDiagnostic() const = 0;
  virtual void
    Set_PacketId( FSP_PacketId id ) = 0;
  virtual void
```

```
    Set_ExpectedPacketId( FSP_PacketId id ) = 0;
  virtual void
    Set_EarliestProdTime( const ISLE_Time& earliestTime ) = 0;
  virtual void
    Put_EarliestProdTime( ISLE_Time* pearliestTime ) = 0;
  virtual void
    Set_LatestProdTime( const ISLE_Time& latestTime ) = 0;
  virtual void
    Put_LatestprodTime( ISLE_Time* platestTime ) = 0;
  virtual void
    Set_DelayTime( SLE_Duration delay ) = 0;
  virtual void
    Set_TransmisionMode( FSP_TransmissionMode mode ) = 0;
  virtual void
    Set_MapId( FSP_MapId id ) = 0;
  virtual void
    Set_Blocking( SLE_YesNo ) = 0;
  virtual void
    Set_ProcessingStartedNotification( SLE_SlduStatusNotification ntf)= 0;
  virtual void
    Set_RadiatedNotification( SLE_SlduStatusNotification ntf )= 0;
  virtual void
    Set_AcknowledgedNotification( SLE_SlduStatusNotification ntf ) = 0;
  virtual void
    Set_Data( size_t length, const SLE_Octet* pdata ) = 0;
  virtual void
    Put_Data( size_t length, SLE_Octet* pdata ) = 0;
  virtual void
    Set_PacketBufferAvailable( FSP_BufferSize bufAvail ) = 0;
  virtual void
    Set_TransferDataDiagnostic( FSP_TransferDataDiagnostic diagnostic)= 0;
};
```

Methods

`FSP_PacketId Get_PacketId() const;`

Returns the packet identification.

`FSP_PacketId Get_ExpectedPacketId() const;`

Returns the next expected packet identification. If the parameter has not been set returns zero.

`const ISLE_Time* Get_EarliestProdTime() const;`

Returns a pointer to the earliest production time, if the parameter has been specified. If the parameter is not set, returns a NULL pointer.

`const ISLE_Time* Get_LatestProdTime() const;`

Returns a pointer to the latest production time, if the parameter has been specified. If the parameter is not set, returns a NULL pointer.

SLE_Duration Get_DelayTime() const;

Returns the parameter delay time.

FSP_TransmissionMode Get_TransmisionMode() const;

Returns the transmission mode parameter.

virtual bool Get_MapIdUsed()

Returns TRUE if the MAP ID parameter is used and not set to 'none'. Otherwise returns FALSE.

FSP_MapId Get_MapId() const;

Returns the MAP identifier if set in the object.

Precondition: `Get_MapIdUsed()` returns TRUE.

SLE_YesNo Get_Blocking() const;

Returns the specification whether packet blocking should be applied.

SLE_SlduStatusNotification
Get_ProcessingStartedNotification() const;

Returns the specification whether a notification shall be sent when processing of the packet was started.

SLE_SlduStatusNotification Get_RadiatedNotification() const;

Returns the specification whether a notification shall be sent when the packet was radiated.

SLE_SlduStatusNotification Get_AcknowledgedNotification() const;

Returns the specification whether a notification shall be sent when the packet was received on board.

const SLE_Octet* Get_Data(size_t& length) const;

Returns a pointer to the packet data in the object. The data must neither be modified nor deleted by the caller.

Arguments

`length` the number of bytes in the packet

SLE_Octet* Remove_Data(size_t& length);

Returns a pointer to the packet data and removes the data from the object. The client is expected to delete the data when they are no longer needed.

Arguments

`length` the number of bytes in the packet

FSP_BufferSize Get_PacketBufferAvailable() const;

Returns the available packet buffer size in bytes if the argument has been set. If the parameter has not been set returns zero.

FSP_TransferDataDiagnostic Get_TransferDataDiagnostic() const;

Returns the diagnostic code.

Precondition: the result is negative, and the diagnostic type is set to 'specific'.

void Set_PacketId(FSP_PacketId id);

Sets the packet identification for the packet transferred.

void Set_ExpectedPacketId(FSP_PacketId id);

Sets the next expected packet identification.

void Set_EarliestProdTime(const ISLE_Time& earliestTime);

Sets the earliest production time to a copy of the input argument.

void Put_EarliestProdTime(ISLE_Time* pearliestTime);

Stores the input argument to the parameter earliest production time.

void Set_LatestProdTime(const ISLE_Time& latestTime);

Sets the latest production time to a copy of the input argument.

void Put_LatestProdTime(ISLE_Time* platestTime);

Stores the input argument to the parameter latest production time.

void Set_DelayTime(SLE_Duration delay);

Sets the parameter delay time.

void Set_TransmisionMode(FSP_TransmissionMode mode);

Sets the parameter transmission mode to the value of the argument.

void Set_MapId(FSP_MapId id);

Sets the parameter 'MAP identifier' to the value of the argument. The argument must be in the range 0 to 63. The method must not be invoked when segment headers are not used.

void Set_Blocking(SLE_YesNo);

Sets the parameter 'blocking' to the value of the argument.

void
Set_ProcessingStartedNotification(SLE_SlduStatusNotification ntf);

Sets the parameter 'processing started notification' to the value of the argument.

void Set_RadiatedNotification(SLE_SlduStatusNotification ntf);

Sets the parameter 'radiated notification' to the value of the argument.

void Set_AcknowledgedNotification(SLE_SlduStatusNotification ntf);

Sets the parameter 'acknowledged notification' to the value of the argument.

void Set_Data(size_t length, const SLE_Octet* pdata);

Copies length bytes from the address `pdata` to the internal packet data parameter.

Arguments

`pdata` pointer to the packet data

`length` the number of bytes in the packet

void Put_Data(size_t length, SLE_Octet* data);

Stores the packet data to the object. The operation object will eventually delete the data buffer.

Arguments

`pdata` pointer to the packet data

`length` the number of bytes in the packet

```
void Set_PacketBufferAvailable( FSP_BufferSize bufAvail );
```

Sets the available packet buffer size in byte.

```
void
Set_TransferDataDiagnostic( FSP_TransferDataDiagnostic diagnostic );
```

Sets the result to 'negative', the diagnostic type to 'specific', and stores the value of the diagnostic code passed by the argument.

Initial Values of Operation Parameters after Creation

Parameter	Created directly	Created by Service Instance
packet identification	0	0
expected packet identification	0	0
earliest production time	NULL	NULL
latest production time	NULL	NULL
delay time	0	0
transmission mode	'invalid'	'invalid'
MAP identifier	(not used)	(not used)
blocking	'invalid'	'invalid'
processing started notification	'invalid'	'invalid'
radiated notification	'invalid'	'invalid'
acknowledged notification	'invalid'	'invalid'
packet buffer available	0	0
transfer buffer diagnostic	'invalid'	'invalid'

Checking of Invocation Parameters

Parameter	Required condition
earliest production time	if earliest and latest production times are set, must be earlier than latest radiation time
latest production time	if earliest and latest production times are set, must be later than earliest radiation time
data	must not be NULL
transmission mode	must not be ‘invalid’
MAP identifier	if used must be a number between 0 and 63 (inclusive)
blocking	must not be ‘invalid’
processing started notification	must not be ‘invalid’
radiated notification	must not be ‘invalid’
acknowledged notification	must not be ‘invalid’; if ‘transmission mode’ is ‘expedited’ must not be ‘produce notification’

Additional Return Codes for `VerifyInvocationArguments`

`SLE_E_TIMERANGE` specification of the earliest and latest production times is inconsistent

Checking of Return Parameters

Parameter	Required condition
expected packet identification	If result is ‘positive’, must be packet identification + 1
transfer buffer diagnostic	must not be ‘invalid’ if the result is ‘negative’ and the diagnostic type is ‘specific’

CCSDS HISTORICAL DOCUMENT

A3.3 FSP ASYNC NOTIFY OPERATION

Name `IFSP_AsyncNotify`
GUID `{91DCEBA1-E896-11d4-9F17-00104B4F22C0}`
Inheritance: `IUnknown - ISLE_Operation`
File `IFSP_AsyncNotify.H`

The interface provides access to the parameters of the unconfirmed operation FSP-ASYNC-NOTIFY.

Synopsis

```
#include <FSP_Types.h>
#include <ISLE_Operation.H>
interface ISLE_Time;
#define IID_IFSP_AsyncNotify_DEF { 0x91dceba1, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_AsyncNotify : ISLE_Operation
{
  virtual FSP_NotificationType
    Get_NotificationType() const = 0;
  virtual FSP_DirectiveId
    Get_DirectiveExecutedId() const = 0;
  virtual FSP_EventInvocationId
    Get_EventThrownId() const = 0;
  virtual const FSP_PacketId*
    Get_PacketIdentificationList( int& size ) const = 0;
  virtual FSP_FopAlert
    Get_FopAlert() const = 0;
  virtual bool
    Get_PacketsProcessed() const = 0;
  virtual FSP_PacketId
    Get_PacketLastProcessed() const = 0;
  virtual const ISLE_Time*
    Get_ProductionStartTime() const = 0;
  virtual FSP_PacketStatus
    Get_PacketStatus() const = 0;
  virtual bool
    Get_PacketsCompleted() const = 0;
  virtual FSP_PacketId
    Get_PacketLastOk() const = 0;
  virtual const ISLE_Time*
    Get_ProductionStopTime() const = 0;
  virtual FSP_ProductionStatus
    Get_ProductionStatus() const = 0;
  virtual void
    Set_NotificationType( FSP_NotificationType notifyType ) = 0;
  virtual void
    Set_DirectiveExecutedId( FSP_DirectiveId id ) = 0;
  virtual void
    Set_EventThrownId( FSP_EventInvocationId id ) = 0;
  virtual void
    Set_PacketIdentificationList( const FSP_PacketId* list,
                                  int size ) = 0;
  virtual void
```


CCSDS HISTORICAL DOCUMENT

```
    Put_PacketIdentificationList( FSP_PacketId* list, int size ) = 0;
  virtual void
    Set_FopAlert( FSP_FopAlert alert) = 0;
  virtual void
    Set_PacketLastProcessed( FSP_PacketId id ) = 0;
  virtual void
    Set_ProductionStartTime( const ISLE_Time& startTime ) = 0;
  virtual void
    Put_ProductionStartTime( ISLE_Time* pstartTime ) = 0;
  virtual void
    Set_PacketStatus( FSP_PacketStatus status ) = 0;
  virtual void
    Set_PacketLastOk( FSP_PacketId id ) = 0;
  virtual void
    Set_ProductionStopTime( const ISLE_Time& stopTime ) = 0;
  virtual void
    Put_ProductionStopTime( ISLE_Time* pstopTime ) = 0;
  virtual void
    Set_ProductionStatus( FSP_ProductionStatus status ) = 0;
};
```

Methods

FSP_NotificationType Get_NotificationType() const;

Returns the notification type.

FSP_DirectiveId Get_DirectiveExecutedId() const;

Returns the identification of the executed directive to which the notification refers.

Precondition: notification type is one of ‘positive confirm response to directive’, or ‘negative confirm response to directive’.

FSP_EventInvocationId Get_EventThrownId() const;

Returns the identification of the thrown event to which the notification refers.

Precondition: notification type is one of ‘action list completed’, ‘action list not completed’, ‘event condition evaluate to false’.

const FSP_PacketId* Get_PacketIdentificationList(int& size) const;

Returns the list of identifiers of affected packets. If the parameter is present but the list is empty, or if the parameter is not present, returns a NULL pointer.

Precondition: notification type is one of ‘packet processing started’, ‘packet radiated’, ‘packet acknowledged’, ‘sldu expired’, ‘packet transmission mode mismatch’, ‘production interrupted’, ‘VC aborted’, or ‘production halted’.

Arguments

`size` number of packet identifiers in the list

`FSP_FopAlert Get_FopAlert() const;`

Returns the FOP Alert parameter.

Precondition: notification type is 'transmission mode capability change' or 'negative confirm response to directive'.

`bool Get_PacketsProcessed() const;`

Returns true if at least one packet has started processing, false otherwise.

`FSP_PacketId Get_PacketLastProcessed() const;`

Returns the identification of the last packet for which processing started.

Precondition: `Get_PacketsProcessed()` returns true.

`const ISLE_Time* Get_ProductionStartTime() const;`

Returns a pointer to the production start time of the last packet processed if the parameter has been set. Otherwise returns a NULL pointer.

Precondition: `Get_PacketsProcessed()` returns true.

`FSP_Status Get_PacketStatus() const;`

Returns the status of the last packet processed.

Precondition: `Get_PacketsProcessed()` returns true.

`bool Get_PacketsCompleted() const;`

Returns true if at least one packet has successfully completed processing (i.e., radiated for BD and acknowledged for AD), false otherwise.

`FSP_PacketId Get_PacketLastOk() const;`

Returns the identification of the last packet that successfully completed processing.

Precondition: `Get_PacketsCompleted()` returns true.

const ISLE_Time* Get_ProductionStopTime() const;

Returns a pointer to the production stop time of the last packet that successfully completed processing if the parameter has been set. Otherwise returns a NULL pointer.

Precondition: `Get_PacketsCompleted()` returns true.

FSP_ProductionStatus Get_ProductionStatus() const;

Returns the current value of the production status.

void Set_NotificationType(FSP_NotificationType notifyType);

Sets the notification type.

void Set_DirectiveExecutedId(FSP_DirectiveId id);

Sets the identification of the executed directive to which the notification refers.

void Set_EventThrownId(FSP_EventInvocationId id);

Sets the identification of the thrown event to which the notification refers.

void
Set_PacketIdentificationList(const FSP_PacketId* list, int size);

Copies the list of packet identifiers passed as argument to the parameter 'packet identification list'. If the list must be supplied for the notification but does not contain any entries, the size argument must be set to zero. In this special case, a NULL pointer can be supplied.

Arguments

`size` number of packet identifiers in the list

void Put_PacketIdentificationList(FSP_PacketId* list, int size);

Stores the list of packet identifiers passed as argument to the parameter 'packet identification list'. If the list must be supplied for the notification but does not contain any entries, the size argument must be set to zero. In this special case, a NULL pointer can be supplied.

Arguments

`size` number of packet identifiers in the list

```
void Set_FopAlert( FSP_FopAlert alert );
```

Sets the parameter FOP alert to the value passed as argument.

```
void Set_PacketLastProcessed( FSP_PacketId id ) = 0;
```

Sets the identification of the last packet processed and sets ‘packets processed’ to true.

```
void Set_ProductionStartTime( const ISLE_Time& startTime );
```

Sets the production start time of the last processed packet to a copy of the input argument.

```
void Put_ProductionStartTime( ISLE_Time* pstartTime );
```

Stores the input argument to the parameter production start time of the packet last processed.

```
void Set_PacketStatus( FSP_Status status );
```

Sets the status of the last processed packet.

```
void Set_PacketLastOk( FSP_PacketId id );
```

Sets the identification of the last packet that completed processing and sets ‘packets completed to true.

```
void Set_ProductionStopTime( const ISLE_Time& stopTime );
```

Sets the radiation stop time of the last packet that completed processing to a copy of the input argument.

```
void Put_ProductionStopTime( ISLE_Time* pstopTime );
```

Stores the input argument to the parameter radiation stop time of the packet last radiated.

```
void Set_ProductionStatus( FSP_ProductionStatus status );
```

Sets the value of the parameter production status.

Initial Values of Operation Parameters after Creation

Parameter	Created directly	Created by Service Instance
notification type	‘invalid’	‘invalid’
directive executed identification	0	0
event thrown identification	0	0
packet identification list	NULL	NULL
FOP alert	‘invalid’	‘invalid’
packets processed	FALSE	TRUE if the number of packets processed is > 0, FALSE otherwise
packet identification last processed	0	value stored for status reports
production start time	NULL (not used)	value stored for status reports
packet status	‘invalid’	value stored for status reports
packets completed	FALSE	TRUE if the number of BD packets radiated is > 0 or the number of AD packets acknowledged > 0, FALSE otherwise
packet identification last OK	0	value stored for status reports
production stop time	NULL (not used)	value stored for status reports
production status	‘invalid’	value stored for status reports

Checking of Invocation Parameters

Parameter	Required condition
notification type	Must not be ‘invalid’.
packet identification list	Must be present and have a single entry for the notification types ‘packet processing started’, ‘packet radiated’, and ‘packet acknowledged’. Must be present with one or more entries for the notification types ‘sldu expired’ and ‘production interrupted’. Must be present with any number of entries (including zero) for the notification types ‘packet transmission mode mismatch’, ‘production interrupted’, ‘VC aborted’, or ‘production halted’. Must not be present for all other notifications.
FOP alert	Must not be ‘invalid’ for the notifications ‘transmission mode capability change’ or ‘negative confirm response to directive’. Must be ‘invalid’ for all other notifications.
packets processed	Must not be FALSE if the notification type is ‘packet processing started’, ‘packet radiated’, ‘packet acknowledged’, ‘sldu expired’, and ‘production interrupted’. Must not be FALSE if ‘packets completed’ is TRUE.
production start time	Must not be NULL if ‘packets processed’ is TRUE
packet status	Must not be ‘invalid’ if ‘packets processed’ is TRUE
packets completed	Must not be FALSE if the notification type is ‘packet acknowledged’.
radiation stop time	Must not be NULL if ‘packets completed’ is TRUE
production status	Must not be ‘invalid.

A3.4 FSP STATUS REPORT OPERATION

Name IFSP_StatusReport
GUID {91DCEBA3-E896-11d4-9F17-00104B4F22C0}
Inheritance: IUnknown – ISLE_Operation
File IFSP_StatusReport.H

The interface provides access to the parameters of the unconfirmed operation FSP-STATUS-REPORT.

Synopsis

```
#include <FSP_Types.h>
#include <ISLE_Operation.H>
interface ISLE_Time;

#define IID_IFSP_StatusReport_DEF { 0x91dceba3, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_StatusReport : ISLE_Operation
{
  virtual bool
    Get_PacketsProcessed() const = 0;
  virtual FSP_PacketId
    Get_PacketLastProcessed() const = 0;
  virtual const ISLE_Time*
    Get_ProductionStartTime() const = 0;
  virtual FSP_PacketStatus
    Get_PacketStatus() const = 0;
  virtual bool
    Get_PacketsCompleted() const = 0;
  virtual FSP_PacketId
    Get_PacketLastOk() const = 0;
  virtual const ISLE_Time*
    Get_ProductionStopTime() const = 0;
  virtual FSP_ProductionStatus
    Get_ProductionStatus() const = 0;
  virtual unsigned long
    Get_NumberOfADPacketsReceived() const = 0;
  virtual unsigned long
    Get_NumberOfBDPacketsReceived() const = 0;
  virtual unsigned long
    Get_NumberOfADPacketsProcessed() const = 0;
  virtual unsigned long
    Get_NumberOfBDPacketsProcessed() const = 0;
  virtual unsigned long
    Get_NumberOfADPacketsRadiated() const = 0;
  virtual unsigned long
    Get_NumberOfBDPacketsRadiated() const = 0;
  virtual unsigned long
    Get_NumberOfPacketsAcknowledged() const = 0;
  virtual FSP_BufferSize
    Get_PacketBufferAvailable() const = 0;
  virtual void
    Set_PacketLastProcessed( FSP_PacketId id ) = 0;
  virtual void
```

```
    Set_ProductionStartTime( const ISLE_Time& startTime ) = 0;
  virtual void
    Put_ProductionStartTime( ISLE_Time* pstartTime ) = 0;
  virtual void
    Set_PacketStatus( FSP_PacketStatus status ) = 0;
  virtual void
    Set_PacketLastOk( FSP_PacketId id ) = 0;
  virtual void
    Set_ProductionStopTime( const ISLE_Time& stopTime ) = 0;
  virtual void
    Put_ProductionStopTime( ISLE_Time* pstopTime ) = 0;
  virtual void
    Set_ProductionStatus( FSP_ProductionStatus status ) = 0;
  virtual void
    Set_NumberOfADPacketsReceived( unsigned long numRecv ) = 0;
  virtual void
    Set_NumberOfBDPacketsReceived( unsigned long numRecv ) = 0;
  virtual void
    Set_NumberOfADPacketsProcessed( unsigned long numProc ) = 0;
  virtual void
    Set_NumberOfBDPacketsProcessed( unsigned long numProc ) = 0;
  virtual void
    Set_NumberOfADPacketsRadiated( unsigned long numRad ) = 0;
  virtual void
    Set_NumberOfBDPacketsRadiated( unsigned long numRad ) = 0;
  virtual void
    Set_NumberOfPacketsAcknowledged( unsigned long numAck ) = 0;
  virtual void
    Set_PacketBufferAvailable( FSP_BufferSize size ) = 0;
};
```

Methods

`bool Get_PacketsProcessed() const;`

Returns true if at least one packet started processing. This condition is true if the number of AD and BD packets processed are not both zero.

`FSP_PacketId Get_PacketLastProcessed() const;`

Returns the identification of the packet last processed.

Precondition: `Get_PacketsProcessed()` returns true.

`const ISLE_Time* Get_ProductionStartTime() const;`

Returns a pointer to the radiation start time of the last packet processed, if the parameter has been set. Otherwise returns a NULL pointer.

FSP_Status Get_PacketStatus() const;

Returns the status of the last packet for which processing started.

Precondition: `Get_PacketsProcessed()` returns true.

bool Get_PacketsCompleted() const;

Returns true if at least one packet successfully completed processing. The condition is true if the number of BD packets radiated and the number of AD packets acknowledged are not both zero.

FSP_PacketId Get_PacketLastOk() const;

Returns the identification of the last packet which successfully completed processing.

Precondition: `Get_PacketsCompleted()` returns true.

const ISLE_Time* Get_ProductionStopTime() const;

Returns a pointer to the production stop time of the last packet that successfully completed processing, if the parameter has been set. Otherwise returns a NULL pointer.

FSP_ProductionStatus Get_ProductionStatus() const;

Returns the current value of the production status.

unsigned long Get_NumberOfADPacketsReceived() const;

Returns the number of AD packets that have been received and accepted by the provider.

unsigned long Get_NumberOfBDPacketsReceived() const;

Returns the number of BD packets that have been received and accepted by the provider.

unsigned long Get_NumberOfADPacketsProcessed() const;

Returns the number of AD packets for which processing was started.

unsigned long Get_NumberOfBDPacketsProcessed() const;

Returns the number of BD packets for which processing was started.

unsigned long Get_NumberOfADPacketsRadiated() const;

Returns the number of AD packets that have been successfully radiated by the provider.

CCSDS HISTORICAL DOCUMENT


```
unsigned long Get_NumberOfBDPacketsRadiated() const;
```

Returns the number of BD packets that have been successfully radiated by the provider.

```
unsigned long Get_NumberOfPacketsAcknowledged() const;
```

Returns the number of packets that have been acknowledged via the CLCW.

```
FSP_BufferSize Get_PacketBufferAvailable() const;
```

Returns the size of the available packet buffer in octets.

```
void Set_ProductionStartTime( const ISLE_Time& startTime );
```

Sets the production start time of the packet last processed to a copy of the input argument.

```
void Put_ProductionStartTime( ISLE_Time* pstartTime );
```

Stores the input argument to the parameter production start time.

```
void Set_PacketStatus( FSP_Status status );
```

Sets the status of the packet last processed.

```
void Set_PacketLastOk( FSP_PacketId id );
```

Sets the identification of the last packet, which successfully completed production.

```
void Set_ProductionStopTime( const ISLE_Time& stopTime );
```

Sets the production stop time of the packet last that successfully completed production to a copy of the input argument.

```
void Put_ProductionStopTime( ISLE_Time* pstopTime );
```

Stores the input argument to the parameter production stop time.

```
void Set_ProductionStatus( FSP_ProductionStatus status );
```

Sets the value of the production status.

```
void Set_NumberOfADPacketsReceived( unsigned long numRecv );
```

Sets the number of AD packets received and accepted by the provider.

CCSDS HISTORICAL DOCUMENT

void Set_NumberOfBDPacketsReceived(unsigned long numRecv);

Sets the number of BD packets received and accepted by the provider.

void Set_NumberOfADPacketsProcessed(unsigned long numProc);

Sets the number of AD packets for which processing was started.

void Set_NumberOfBDPacketsProcessed(unsigned long numProc);

Sets the number of BD packets for which processing was started.

void Set_NumberOfADPacketsRadiated(unsigned long numRad);

Sets the number of AD packets successfully radiated by the provider.

void Set_NumberOfBDPacketsRadiated(unsigned long numRad);

Sets the number of BD packets successfully radiated by the provider.

void Set_NumberOfPacketsAcknowledged(unsigned long numRad);

Sets the number of packets acknowledged via the CLCW.

void Set_PacketBufferAvailable(FSP_BufferSize size);

Sets the available buffer size.

Initial Values of Operation Parameters after Creation

The interface `ISLE_SIOpFactory` does not support creation of status report operation objects, as this operation is handled by the service instance internally.

Parameter	Created directly
packet identification last processed	0
production start time	NULL (not used)
packet status	‘invalid’
packet identification last OK	0
production stop time	NULL (not used)
production status	‘invalid’
number of AD packets received	0
number of BD packets received	0
number of AD packets processed	0
number of BD packets processed	0
number of AD packets radiated	0
number of AD packets radiated	0
number of AD packets acknowledged	0
packet buffer available	0

Checking of Invocation Parameters

Parameter	Required condition
production start time	Must not be NULL if number of AD packets processed > 0 OR number of BD packets processed > 0
packet status	Must not be ‘invalid’ if number of AD packets processed > 0 OR number of BD packets processed > 0
production stop time	Must not be NULL if number of BD packets radiated > 0 OR number of AD packets acknowledged > 0
production status	Must not be ‘invalid’
number of packets AD received	Must be ≥ number of AD packets processed
number of packets BD received	Must be ≥ number of BD packets processed
number of AD packets processed	Must be ≥ number of AD packets radiated and ≤ number of packets AD received
number of BD packets processed	Must be ≥ number of BD packets radiated and ≤ number of packets BD received
number of AD packets radiated	Must be ≤ number of AD packets processed
number of BD packets radiated	Must be ≤ number of BD packets processed
number of AD packets acknowledged	Must be ≤ number of AD packets radiated

A3.5 FSP GET PARAMETER OPERATION

Name `IFSP_GetParameter`
GUID `{91DCEBA4-E896-11d4-9F17-00104B4F22C0}`
Inheritance: `IUnknown – ISLE_Operation – ISLE_ConfirmedOperation`
File `IFSP_GetParameter.H`

The interface provides access to the parameters of the confirmed operation FSP-GET-PARAMETER.

Synopsis

```
#include <FSP_Types.h>
#include <ISLE_ConfirmedOperation.H>

#define IID_IFSP_GetParameter_DEF { 0x91dceba4, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_GetParameter : ISLE_ConfirmedOperation
{
  virtual FSP_ParameterName
    Get_RequestedParameter() const = 0;
  virtual FSP_ParameterName
    Get_ReturnedParameter() const = 0;
  virtual const FSP_ApId*
    Get_ApIdList( int& size ) const = 0;
  virtual unsigned long
    Get_BlockingTimeout() const = 0;
  virtual FSP_BlockingUsage
    Get_BlockingUsage() const = 0;
  virtual SLE_DeliveryMode
    Get_DeliveryMode() const = 0;
  virtual SLE_YesNo
    Get_DirectiveInvocationEnabled() const = 0;
  Virtual SLE_YesNo
    Get_DirectiveInvocationOnline() const = 0;
  virtual FSP_DirectiveId
    Get_ExpectedDirectiveId() const = 0;
  virtual FSP_EventInvocationId
    Get_ExpectedEventInvocationId() const = 0;
  virtual FSP_PacketId
    Get_ExpectedSlduId() const = 0;
  virtual unsigned long
    Get_FopSlidingWindow() const = 0;
  virtual FSP_FopState
    Get_FopState() const = 0;
  virtual const FSP_MapId*
    Get_MapList( int& size ) const = 0;
  virtual const FSP_AbsolutePriority*
    Get_MapPriorityList( int& size ) const = 0;
  virtual const FSP_MapId*
    Get_MapPollingVector( int& size ) const = 0;
  virtual FSP_MuxScheme
    Get_MapMuxScheme() const = 0;
  virtual unsigned long
    Get_MaxFrameLength() const = 0;
```

```
virtual unsigned long
  Get_MaxPacketLength() const = 0;
virtual FSP_PermittedTransmissionMode
  Get_PermittedTransmissionMode() const = 0;
virtual unsigned long
  Get_ReportingCycle() const = 0;
virtual unsigned long
  Get_ReturnTimeoutPeriod() const = 0;
virtual SLE_YesNo
  Get_SegmentHeaderPresent() const = 0;
virtual FSP_TimeoutType
  Get_TimeoutType() const = 0;
virtual unsigned long
  Get_TimerInitial() const = 0;
virtual unsigned long
  Get_TransmissionLimit() const = 0;
virtual unsigned long
  Get_TransmitterFrameSequenceNumber() const = 0;
virtual const FSP_AbsolutePriority*
  Get_VcPriorityList( int& size ) const = 0;
virtual const FSP_VcId*
  Get_VcPollingVector( int& size ) const = 0;
virtual FSP_MuxScheme
  Get_VcMuxScheme() const = 0;
virtual FSP_VcId
  Get_VirtualChannel() const = 0;
virtual FSP_GetParameterDiagnostic
  Get_GetParameterDiagnostic() const = 0;
virtual void
  Set_RequestedParameter( FSP_ParameterName name ) = 0;
virtual void
  Set_ApIdList( const FSP_ApId* plist,
                int size ) = 0;
virtual void
  Put_ApIdList( FSP_ApId* plist,
                int size ) = 0;
virtual void
  Set_BlockingTimeout( unsigned long timeout ) = 0;
virtual void
  Set_BlockingUsage( FSP_BlockingUsage usage ) = 0;
virtual void
  Set_DeliveryMode() = 0;
virtual void
  Set_DirectiveInvocationEnabled( SLE_YesNo yesNo ) = 0;
virtual void
  Set_DirectiveInvocationOnline( SLE_YesNo yesNo )
virtual void
  Set_ExpectedDirectiveId( FSP_DirectiveId id) = 0;
virtual void
  Set_ExpectedEventInvocationId( FSP_EventInvocationId id) = 0;
virtual void
  Set_ExpectedSlduId( FSP_PacketId id ) = 0;
virtual void
  Set_FopSlidingWindow( unsigned long window ) = 0;
virtual void
  Set_FopState( FSP_FopState state ) = 0;
virtual void
  Set_MapList( const FSP_MapId* plist,
               int size ) = 0;
virtual void
```

```
    Put_MapList( FSP_MapId* plist,
                 int size ) = 0;
  virtual void
    Set_MapPriorityList( const FSP_AbsolutePriority* priorities,
                         int size ) = 0;
  virtual void
    Put_MapPriorityList( FSP_AbsolutePriority* priorities,
                         int size ) = 0;
  virtual void
    Set_MapPollingVector( const FSP_MapId* pvec,
                          int size ) = 0;
  virtual void
    Put_MapPollingVector( FSP_MapId* pvec,
                          int size ) = 0;
  virtual void
    Set_MapMuxScheme( FSP_MuxScheme scheme ) = 0;
  virtual void
    Set_MaxFrameLength( unsigned long length ) = 0;
  virtual void
    Set_MaxPacketLength( unsigned long length ) = 0;
  virtual void
    Set_PermittedTransmissionMode( FSP_PermittedTransmissionMode mode) = 0;
  virtual void
    Set_ReportingCycle(unsigned long cycle ) = 0;
  virtual void
    Set_ReturnTimeoutPeriod( unsigned long period) = 0;
  virtual void
    Set_SegmentHeaderPresent( SLE_YesNo yesNo ) = 0;
  virtual void
    Set_TimeoutType( FSP_TimeoutType type ) = 0;
  virtual void
    Set_TimerInitial( unsigned long timeout ) = 0;
  virtual void
    Set_TransmissionLimit( unsigned long limit ) = 0;
  virtual void
    Set_TransmitterFrameSequenceNumber( unsigned long number ) = 0;
  virtual void
    Set_VcPriorityList( const FSP_AbsolutePriority* priorities,
                        int size ) = 0;
  virtual void
    Put_VcPriorityList( FSP_AbsolutePriority* priorities,
                        int size ) = 0;
  virtual void
    Set_VcPollingVector( const FSP_VcId* pvec,
                         int size ) = 0;
  virtual void
    Put_VcPollingVector( FSP_VcId* pvec,
                         int size ) = 0;
  virtual void
    Set_VcMuxScheme( FSP_MuxScheme scheme ) = 0;
  virtual void
    Set_VirtualChannel( FSP_VcId id ) = 0;
  virtual void
    Set_GetParameterDiagnostic
    ( FSP_GetParameterDiagnostic diagnostic ) = 0;
};
```

Methods

`FSP_ParameterName Get_RequestedParameter() const;`

Returns the parameter for which the value shall be reported.

`FSP_ParameterName Get_ReturnedParameter() const;`

Returns the parameter for which the value is reported. Following the return, this must be identical to the result of `Get_RequestedParameter()`.

`const FSP_ApId* Get_ApIdList( int& size ) const;`

Returns an array of Application Process Identifiers to which the service instance has access, or NULL if any API may be accessed.

Precondition: the returned parameter is `apid-list`.

Arguments

`size` set to the number of identifiers in the list (zero, if no list is supplied)

`unsigned long Get_BlockingTimeout() const;`

Returns timeout period in milliseconds for blocking of packets. If blocking is not used, returns zero.

Precondition: the returned parameter is `blocking-timeout-period`.

`FSP_BlockingUsage Get_BlockingUsage() const;`

Returns a specification whether blocking of packets is permitted.

Precondition: the returned parameter is `blocking-usage`.

`SLE_DeliveryMode Get_DeliveryMode() const;`

Returns ‘forward online’.

Precondition: the returned parameter is `delivery-mode`.

`SLE_YesNo Get_DirectiveInvocationEnabled() const;`

Returns whether this service instance is authorized to invoke the FSP-INVOKE-DIRECTIVE operation.

Precondition: the returned parameter is `directive-invocation-enabled`.

`SLE_YesNo Get_DirectiveInvocationOnline() const;`

Returns ‘yes’ if a service instance with directive invocation capability is currently bound to the service provider and ‘no’ otherwise.

Precondition: the returned parameter is `directive-invocation-online`.

`FSP_DirectiveId Get_ExpectedDirectiveId() const;`

Returns the next expected directive identification.

Precondition: the returned parameter is `expected-directive-identification`.

`FSP_EventInvocationId Get_ExpectedEventInvocationId() const;`

Returns the next expected event invocation identifier.

Precondition: the returned parameter is `expected-event-invocation-identification`.

`FSP_PacketId Get_ExpectedSlduId() const;`

Returns the next expected packet identification.

Precondition: the returned parameter is `expected-sldu-identification` and the value has been set via a START invocation or as result of a TRANSFER DATA operation.

`unsigned long Get_FopSlidingWindow() const;`

Returns the width of the FOP sliding window.

Precondition: the returned parameter is `fop-sliding-window`.

`FSP_FopState Get_FopState() const;`

Returns the state of the FOP.

Precondition: the returned parameter is `fop-state`.

`const FSP_MapId* Get_MapList( int& size ) const;`

Returns an array of MAP identifiers for the MAPs that can be used by the service instance. If no MAPs are used returns a NULL pointer.

Precondition: the returned parameter is `map-list`.

Arguments

`size` set to the number of identifiers in the list

`const FSP_AbsolutePriority* Get_MapPriorityList(int& size) const;`

Returns the priority specification for multiplexing on MAPs if the multiplexing scheme is 'absolute priority'. The priority specification is an array of 'map id' - 'priority' pairs. If the multiplexing scheme is FIFO or 'polling vector', or if MAPs are not used, returns NULL.

Precondition: the returned parameter is `map-multiplexing-control`.

Arguments

`size` set to the number of 'map id' - 'priority' pairs in the list

`const FSP_MapId* Get_MapPollingVector( int& size ) const;`

Returns the polling vector for multiplexing on MAPs if the multiplexing scheme is 'polling vector'. The polling vector is an array of MAP identifiers. If the multiplexing scheme is FIFO or 'absolute priority', or if MAPs are not used, returns NULL.

Precondition: the returned parameter is `map-multiplexing-control`.

Arguments

`size` set to the number of identifiers in the list

`FSP_MuxScheme Get_MapMuxScheme() const;`

Returns the multiplexing scheme in effect for MAPs. If MAPs are not used, the parameter is set to 'invalid'.

Precondition: the returned parameter is `map-multiplexing-scheme`.

`unsigned long Get_MaxFrameLength() const;`

Returns the maximum length of a TC frame in octets.

Precondition: the returned parameter is `maximum-frame-length`.

`unsigned long Get_MaxPacketLength() const;`

Returns the maximum length of a packet in octets.

Precondition: the returned parameter is `maximum-packet-length`.

FSP_PermittedTransmissionMode Get_PermittedTransmissionMode() const;

Returns the permitted transmission mode.

Precondition: the returned parameter is `permitted-transmission-mode`.

unsigned long GetReportingCycle() const;

Returns the reporting cycle requested by the user if periodic reporting is active. If reporting is not active, returns zero.

Precondition: the returned parameter is `reporting-cycle`.

unsigned long Get_ReturnTimeoutPeriod() const;

Returns the return timeout period used by the provider.

Precondition: the returned parameter is `return-timeout-period`.

SLE_YesNo Get_SegmentHeaderPresent() const;

Returns whether segment headers are used.

Precondition: the returned parameter is `segment-header`.

FSP_TimeoutType Get_TimeoutType() const;

Returns the FOP timeout type parameter, which specifies how the FOP reacts when the maximum number of retransmissions has been exceeded.

Precondition: the returned parameter is `timeout-type`.

unsigned long Get_TimerInitial() const;

Returns the initial value of the countdown timer in microseconds when an AD or BC frame is transmitted.

Precondition: the returned parameter is `timer-initial`.

unsigned long Get_TransmissionLimit() const;

Returns the maximum number of times the first frame on the Sent-Queue may be transmitted.

Precondition: the returned parameter is `transmission-limit`.

unsigned long Get_TransmitterFrameSequenceNumber() const;

Returns the Transmitter Frame Sequence Number, V(S), which contains the value of the Frame Sequence Number, N(S), to be put in the Transfer Frame Header of the next Type-AD frame to be transmitted.

Precondition: the returned parameter is `transmitter-frame-sequence-number`.

const FSP_AbsolutePriority* Get_VcPriorityList(int& size) const;

Returns the priority specification for multiplexing on VCs if the multiplexing scheme is ‘absolute priority’. The priority specification is an array of ‘VC ID’ - ‘priority’ pairs. If the multiplexing scheme is FIFO or ‘polling vector’, returns NULL.

Precondition: the returned parameter is `vc-multiplexing-control`.

Arguments

`size` set to the number of ‘VC ID’ - ‘priority’ pairs in the list

const FSP_VcId* Get_VcPollingVector(int& size) const;

Returns the polling vector for multiplexing on VCs if the multiplexing scheme is ‘polling vector’. The polling vector is an array of VC IDs. If the multiplexing scheme is FIFO or ‘absolute priority’, returns NULL.

Precondition: the returned parameter is `vc-multiplexing-control`.

Arguments

`size` set to the number of identifiers in the list

FSP_MuxScheme Get_VcMuxScheme() const;

Returns the multiplexing scheme in effect for VCs.

Precondition: the returned parameter is `vc-multiplexing-scheme`.

FSP_VcId Get_VirtualChannel() const;

Returns the VC being used by this service instance.

Precondition: the returned parameter is `virtual-channel`.

`FSP_GetParameterDiagnostic Get_GetParameterDiagnostic() const;`

Returns the diagnostic code.

<u>Precondition</u>: the result is negative, and the diagnostic type is set to 'specific'.

`void Set_RequestedParameter( FSP_ParameterName name );`

Sets the parameter for which the provider shall report the value.

`void Set_ApIdList( const FSP_ApId* plist, int size );`

Sets the returned parameter name to `apid-list` and copies the list supplied by the arguments to that parameter. The parameter `plist` may be set to NULL to indicate that any APID can be accessed.

<u>Arguments</u>

`plist` array of application process identifiers where each APID must be in the range 0 to 2047, or NULL

`size` number of identifiers in the array

`void Put_ApIdList( FSP_ApId* plist, int size );`

Sets the returned parameter name to `apid-list` and stores the list supplied by the arguments to that parameter. The parameter `plist` may be set to NULL to indicate that any APID can be accessed.

<u>Arguments</u>

`plist` array of application process identifiers where each APID must be in the range 0 to 2047 or, NULL

`size` number of identifiers in the array

`void Set_BlockingTimeout( unsigned long timeout );`

Sets the returned parameter name to `blocking-timeout-period` and sets its value as defined by the argument.

`void Set_BlockingUsage( FSP_BlockingUsage usage );`

Sets the returned parameter name to `blocking-usage` and sets its value as defined by the argument.

void Set_DeliveryMode();

Sets the returned parameter name to `delivery-mode` and sets its value to 'fwd online'.

void Set_DirectiveInvocationEnabled(SLE_YesNo yesNo);

Specifies whether directive invocation is enabled for the service instance.

void Set_DirectiveInvocationOnline(SLE_YesNo yesNo)

Specifies whether a service instance with directive invocation for the VC enabled is currently bound to the service provider.

void Set_ExpectedDirectiveId(FSP_DirectiveId id);

Sets the returned parameter name to `expected-directive-id` and sets its value as defined by the argument.

void Set_ExpectedEventInvocationId(FSP_EventInvocationId id);

Sets the returned parameter name to `expected-event-invocation-id` and sets its value as defined by the argument.

void Set_ExpectedSlduId(FSP_PacketId id);

Sets the returned parameter name to `expected-sldu-identification` and sets its value as defined by the argument.

void Set_FopSlidingWindow(unsigned long window);

Sets the returned parameter name to 'fop-sliding-window' and sets its value as defined by the argument.

void Set_FopState(FSP_FopState state);

Sets the returned parameter name to `fop-state` and sets its value as defined by the argument.

void Set_MapList(const FSP_MapId* plist, int size);

Sets the returned parameter name to `map-list` and copies the list supplied by the arguments to that parameter.

Arguments

`plist` array of MAP IDs (each MAP ID must be in the range 0 to 63)

`size` number of identifiers in the array

```
void Put_MapList( FSP_MapId* plist, int size );
```

Sets the returned parameter name to `map-list` and stores the list supplied by the arguments to that parameter.

Arguments

`plist` array of MAP IDs (each MAP ID must be in the range 0 to 63)

`size` number of identifiers in the array

```
void
Set_MapPriorityList( const FSP_AbsolutePriority* priorities,
                     int size);
```

Sets the returned parameter name to `map-multiplexing-control` and copies the list supplied by the arguments to that parameter. Clears the map polling vector if that is set in the object. This method must be used when the multiplexing scheme is 'absolute priority'.

Arguments

`plist` array of MAP ID / Priority pairs as defined by `FSP_AbsolutePriority` (each MAP ID must be in the range 0 to 63 and each priority in the range 1 to 64)

`size` number of structures in the array

```
void Put_MapPriorityList( FSP_AbsolutePriority* priorities,
                          int size );
```

Sets the returned parameter name to `map-multiplexing-control` and stores the list supplied by the arguments to that parameter. Clears the map polling vector if that is set in the object. This method must be used when the multiplexing scheme is 'absolute priority'.

Arguments

`plist` array of MAP ID / Priority pairs as defined by `FSP_AbsolutePriority` (each MAP ID must be in the range 0 to 63 and each priority in the range 1 to 64)

`size` number of structures in the array

void Set_MapPollingVector(const FSP_MapId* pvec, int size);

Sets the returned parameter name to `map-multiplexing-control` and copies the list supplied by the arguments to that parameter. Clears the map priority list if that is set in the object. This method must be used when the multiplexing scheme is 'polling vector'.

<u>Arguments</u>

`pvec` array of MAP IDs (each MAP ID must be in the range 0 to 63)

`size` number of identifiers in the array

void Put_MapPollingVector(FSP_MapId* pvec, int size);

Sets the returned parameter name to `map-multiplexing-control` and stores the list supplied by the arguments to that parameter. Clears the map priority list if that is set in the object. This method must be used when the multiplexing scheme is 'polling vector'.

<u>Arguments</u>

`pvec` array of MAP IDs (each MAP ID must be in the range 0 to 63)

`size` number of identifiers in the array

void Set_MapMuxScheme(FSP_MuxScheme scheme);

Sets the returned parameter name to `map-multiplexing-scheme` and sets its value as defined by the argument.

void Set_MaxFrameLength(unsigned long length);

Sets the returned parameter name to `maximum-frame-length` and sets its value as defined by the argument.

void Set_MaxPacketLength(unsigned long length);

Sets the returned parameter name to `maximum-packet-length` and sets its value as defined by the argument.

void
Set_PermittedTransmissionMode(FSP_PermittedTransmissionMode mode);

Sets the returned parameter name to `permitted-transmission-mode` and sets its value as defined by the argument.

`void Set_ReportingCycle(unsigned long cycle );`

Sets the returned parameter name to `reporting-cycle` and sets its value as defined by the argument.

`void Set_ReturnTimeoutPeriod( unsigned long period);`

Sets the returned parameter name to `return-timeout-period` and sets its value as defined by the argument.

`void Set_SegmentHeaderPresent( SLE_YesNo yesNo );`

Sets the returned parameter name to `segment-header` and sets its value as defined by the argument.

`void Set_TimeoutType( FSP_TimeoutType type );`

Sets the returned parameter name to `timeout-type` and sets its value as defined by the argument.

`void Set_TimerInitial( unsigned long timeout );`

Sets the returned parameter name to `timer-initial` and sets its value as defined by the argument.

Arguments

`timeout` the initial value of the FOP countdown timer in microseconds

`void Set_TransmissionLimit( unsigned long limit );`

Sets the returned parameter name to `transmission-limit` and sets its value as defined by the argument.

Arguments

`limit` the maximum number a frame may be transmitted in the range 1 to 255

`void Set_TransmitterFrameSequenceNumber( unsigned long number );`

Sets the returned parameter name to `transmitter-frame-sequence-number` and sets its value as defined by the argument.

Arguments

`number` the current value of the Transmitter Frame Sequence Number, V(S), which contains the value of the Frame Sequence Number, N(S), to be put in the Transfer Frame Header of the next Type-AD frame to be transmitted. The value must be in the range 0 to 255.

```
void
Set_VcPriorityList(const FSP_AbsolutePriority* priorities,int size);
```

Sets the returned parameter name to `vc-multiplexing-control` and copies the list supplied by the arguments to that parameter. Clears the VC polling vector if that is set in the object. This method must be used when the multiplexing scheme is ‘absolute priority’.

Arguments

`plist` array of VC ID / Priority pairs as defined by `FSP_AbsolutePriority` (each VC ID must be in the range 0 to 63 and each priority in the range 1 to 64)

`size` number of structures in the array

```
void
Put_VcPriorityList( FSP_AbsolutePriority* priorities,
                    int size );
```

Sets the returned parameter name to `vc-multiplexing-control` and stores the list supplied by the arguments to that parameter. Clears the VC polling vector if that is set in the object. This method must be used when the multiplexing scheme is ‘absolute priority’.

Arguments

`plist` array of VC ID / Priority pairs as defined by `FSP_AbsolutePriority` (each VC ID must be in the range 0 to 63 and each priority in the range 1 to 64)

`size` number of structures in the array

```
void Set_VcPollingVector( const FSP_VcId* pvec, int size );
```

Sets the returned parameter name to `vc-multiplexing-control` and copies the list supplied by the arguments to that parameter. Clears the VC priority list if that is set in the object. This method must be used when the multiplexing scheme is ‘polling vector’.

Arguments

`pvec` array of VC IDs (each VC ID must be in the range 0 to 63)

`size` number of identifiers in the array

void Put_VcPollingVector(FSP_VcId* pvec, int size);

Sets the returned parameter name to vc-multiplexing-control and stores the list supplied by the arguments to that parameter. Clears the VC priority list if that is set in the object. This method must be used when the multiplexing scheme is 'polling vector'.

Arguments

pvec array of VC IDs (each VC ID must be in the range 0 to 63)

size number of identifiers in the array

void Set_VcMuxScheme(FSP_MuxScheme scheme);

Sets the returned parameter name to vc-multiplexing-scheme and sets its value as defined by the argument.

void Set_VirtualChannel(FSP_VcId id);

Sets the returned parameter name to virtual-channel and sets its value as defined by the argument.

Arguments

id the value if the VC ID used by the service instance

void
Set_GetParameterDiagnostic(FSP_GetParameterDiagnostic diagnostic);

Sets the result to 'negative', the diagnostic type to 'specific', and stores the value of the diagnostic code passed by the argument.

Initial Values of Operation Parameters after Creation

Parameter	Created directly	Created by Service Instance
requested parameter	'invalid'	'invalid'
returned parameter	'invalid'	'invalid'
apid list	NULL	NULL
blocking timeout period	0	0
blocking usage	'invalid'	'invalid'
directive invocation enabled	'invalid'	'invalid'
directive invocation online	'invalid'	'invalid'
expected sldu identification	0	0
expected event invocation identification	0	0
expected directive identification	0	0
fop-sliding-window	0	0
fop-state	'invalid'	'invalid'
map list	NULL	NULL
map multiplexing control (priority list)	NULL	NULL
map-multiplexing-control (polling vector)	NULL	NULL
map-multiplexing-scheme	'invalid'	'invalid'
maximum frame length	0	0
maximum packet length	0	0
permitted transmission mode	'invalid'	'invalid'
reporting cycle	0	0
return timeout period	0	0
segment header	'invalid'	'invalid'
timeout-type	'invalid'	'invalid'
timer-initial	0	0
transmission-limit	0	0
transmitter-frame-sequence-number	0	0
vc multiplexing control (priority list)	NULL	NULL
vc multiplexing scheme (polling vector)	NULL	NULL
virtual channel	0	0
GET PARAMETER diagnostic	'invalid'	'invalid'

Checking of Invocation Parameters

Parameter	Required condition
requested parameter	must not be 'invalid'

Checking of Return Parameters

The interface ensures consistency between the returned parameter name and the parameter value, as the client cannot set the returned parameter name. The consistency checks defined below only need to be performed when the return is received by the service user. The method `VerifyReturnArguments()` might nevertheless be called on the provider side to check the permissible range of parameter arguments, unless the service element ensures that all values are within the range specified.

Parameter	Required condition
returned parameter	must be the same as the requested parameter
apid list	if not NULL; each element in the list must be in the range 0 to 2047; if the returned parameter is 'apid list', a NULL value indicates that any APID may be accessed.
blocking timeout period	if the returned parameter is 'blocking timeout period' must be either a value between 100 and 100,000 or must be zero (blocking off).
blocking usage	if the returned parameter is 'blocking usage' must not be 'invalid'
directive invocation enabled	if the returned parameter is 'directive invocation enabled' must not be 'invalid'
fop-sliding-window	if the returned parameter is 'fop-sliding-window' must be in the range 1 to 255
fop-state	if the returned parameter is 'fop-state' must not be 'invalid'
map list	if the returned parameter is 'map list' must be either NULL or must contain 1 to 64 MAP identifiers. Each MAP ID must be in the range 0 to 63.
map-multiplexing-control (priority list)	if the returned parameter is 'map-multiplexing-control' must be one of the following: NULL (scheme FIFO or polling vector); A list of 1 to 64 pairs of 'MAP ID' - 'Priority'; The value of the MAP ID must be in the range 0 to 63 and the priority must be in the range 1 to 64;
map-multiplexing-control (polling vector)	if the returned parameter is 'map-multiplexing-control' must be one of the following: NULL (scheme FIFO or absolute priority); A list of 1 to 192 MAP IDs. Each MAP ID must be in the range 0 to 63.
map-multiplexing-scheme	if the returned parameter is 'map-multiplexing-scheme' must not be 'invalid'
maximum frame length	if the returned parameter is 'maximum frame length' must be in the range 12 to 1024.

Parameter	**Required condition**
maximum packet length	if the returned parameter is 'maximum packet length' must be in the range 7 to 65542.
permitted transmission mode	if the returned parameter is 'permitted transmission mode' must not be 'invalid'
return timeout period	if the returned parameter is 'return timeout period 'must not be 0
segment header	if the returned parameter is 'segment header' must not be 'invalid'
timeout-type	if the returned parameter is 'timeout-type' must not be 'invalid'
timer-initial	if the returned parameter is 'timer-initial' must not be zero.
transmission-limit	if the returned parameter is 'transmission-limit' must be in the range 1 to 255.
transmitter-frame-sequence-number	if the returned parameter is 'transmitter-frame-sequence-number' must be in the range 0 to 255
vc multiplexing control (priority scheme)	if the returned parameter is 'vc multiplexing control' must be one of the following: NULL (scheme FIFO or polling vector); A list of 1 to 64 pairs of 'VC ID' - 'Priority'; The value of VC ID must be in the range 0 to 63 and the priority must be in the range 1 to 64;
vc multiplexing control (polling vector)	if the returned parameter is 'vc multiplexing control' must be one of the following: NULL (scheme FIFO or absolute priority); A list of 1 to 192 VC IDs. Each VC ID must be in the range 0 to 63.
vc multiplexing scheme	if the returned parameter is 'vc multiplexing scheme' must not be 'invalid'
virtual channel	if the returned parameter is 'virtual channel' must be in the range 0 to 63.
GET PARAMETER diagnostic	must not be 'invalid' if the result is 'negative' and the diagnostic type is 'specific'

A3.6 FSP THROW EVENT OPERATION

Name `IFSP_ThrowEvent`
GUID `{91DCEBA5-E896-11d4-9F17-00104B4F22C0}`
Inheritance: `IUnknown – ISLE_Operation – ISLE_ConfirmedOperation`
File `IFSP_ThrowEvent.H`

The interface provides access to the parameters of the confirmed operation FSP-THROW-EVENT.

Synopsis

```
#include <FSP_Types.h>
#include <ISLE_ConfirmedOperation.H>

#define IID_IFSP_ThrowEvent_DEF { 0x91dceba5, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_ThrowEvent : ISLE_ConfirmedOperation
{
  virtual unsigned short
    Get_EventId() const = 0;
  virtual const SLE_Octet*
    Get_EventQualifier( size_t& size ) const = 0;
  virtual FSP_EventInvocationId
    Get_EventInvocationId() const = 0;
  virtual FSP_EventInvocationId
    Get_ExpectedEventInvocationId() const = 0;
  virtual FSP_ThrowEventDiagnostic
    Get_ThrowEventDiagnostic() const = 0;
  virtual void
    Set_EventId( unsigned short id ) = 0;
  virtual void
    Set_EventQualifier( size_t size, const SLE_Octet* parg) = 0;
  virtual void
    Set_EventInvocationId( FSP_EventInvocationId id ) = 0;
  virtual void
    Set_ExpectedEventInvocationId( FSP_EventInvocationId id ) = 0;
  virtual void
    Set_ThrowEventDiagnostic (FSP_ThrowEventDiagnostic diagnostic) = 0;
};
```

Methods

`unsigned short Get_EventId() const;`

Returns the identification of the event.

`const SLE_Octet* Get_EventQualifier( size_t& size ) const;`

Returns the event qualifier as an array of octets or NULL if the parameter has not been set in the object.

Arguments

`size` set to the number of octets in the parameter on return

`FSP_EventInvocationId Get_EventInvocationId() const;`

Returns the invocation identifier of the event.

`FSP_EventInvocationId Get_ExpectedEventInvocationId() const;`

Returns the next expected invocation identifier of the event in the return.

`FSP_ThrowEventDiagnostic Get_ThrowEventDiagnostic() const;`

Returns the diagnostic code.

Precondition: the result is negative, and the diagnostic type is set to ‘specific’.

`void Set_EventId( unsigned short id );`

Sets the identifier of the event.

`void Set_EventQualifier( size_t size, const SLE_Octet* parg);`

Copies the octet string passed as argument to the parameter ‘event qualifier’.

Arguments

`parg` pointer to the octet string

`size` the number of octets in the parameter

`void Set_EventInvocationId( FSP_EventInvocationId id );`

Sets the invocation identifier for the event in the invocation.

`void Set_ExpectedEventInvocationId( FSP_EventInvocationId id );`

Sets the next expected invocation identifier for the event in the return.

`void Set_ThrowEventDiagnostic(FSP_ThrowEventDiagnostic diagnostic);`

Sets the result to ‘negative’, the diagnostic type to ‘specific’, and stores the value of the diagnostic code passed by the argument.

Initial Values of Operation Parameters after Creation

Parameter	Created directly	Created by Service Instance
event identifier	0	0
event argument	NULL	NULL
event invocation identifier	0	0
expected event invocation id	0	0
THROW EVENT diagnostic	‘invalid’	‘invalid’

Checking of Invocation Parameters

Parameter	Required condition
event qualifier	must not be NULL
event qualifier length	must be within limits (1 ..128)

Checking of Return Parameters

Parameter	Required condition
THROW EVENT diagnostic	must not be ‘invalid’ if the result is ‘negative’ and the diagnostic type is ‘specific’
expected event invocation id	If result is ‘positive’, must be event invocation id + 1

A3.7 FSP INVOKE DIRECTIVE OPERATION

Name IFSP_InvokeDirective
GUID {91DCEBA8-E896-11d4-9F17-00104B4F22C0}
Inheritance: IUnknown – ISLE_Operation – ISLE_ConfirmedOperation
File IFSP_ThrowEvent.H

The interface provides access to the parameters of the confirmed operation FSP-INVOKE-DIRECTIVE.

Synopsis

```
#include <FSP_Types.h>
#include <ISLE_ConfirmedOperation.H>

#define IID_IFSP_InvokeDirective_DEF { 0x91dceba8, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_InvokeDirective : ISLE_ConfirmedOperation
{
  virtual FSP_DirectiveId
    Get_DirectiveId() const = 0;
  virtual FSP_DirectiveId
    Get_ExpectedDirectiveId() const = 0;
  virtual FSP_Directive
    Get_Directive() const = 0;
  virtual unsigned long
    Get_VR() const = 0;
  virtual unsigned long
    Get_VS() const = 0;
  virtual unsigned long
    Get_FopSlidingWindowWidth() const = 0;
  virtual unsigned long
    Get_TimerInitial() const = 0;
  virtual unsigned long
    Get_TransmissionLimit() const = 0;
  virtual FSP_DirectiveTimeoutType
    Get_TimeoutType() const = 0;
  virtual FSP_AbsolutePriority*
    Get_Priority( int& size ) const = 0;
  virtual FSP_MapId*
    Get_PollingVector( int& size ) const = 0;
  virtual FSP_InvokeDirectiveDiagnostic
    Get_InvokeDirectiveDiagnostic() const = 0;
  virtual void
    Set_DirectiveId( FSP_DirectiveId id ) = 0;
  virtual void
    Set_ExpectedDirectiveId( FSP_DirectiveId id ) = 0;
  virtual void
    Set_InitiateADwithoutCLCW() = 0;
  virtual void
    Set_InitiateADwithCLCW() = 0;
  virtual void
    Set_InitiateADwithUnlock() = 0;
  virtual void
    Set_InitiateADwithSetVR( unsigned long vr ) = 0;
```

```
  virtual void
    Set_TerminateAD() = 0;
  virtual void
    Set_ResumeAD() = 0;
  virtual void
    Set_VS( unsigned long vs ) = 0;
  virtual void
    Set_FopSlidingWindow( unsigned long width ) = 0;
  virtual void
    Set_TimerInitial( unsigned long timeout ) = 0;
  virtual void
    Set_TransmissionLimit( unsigned long limit ) = 0;
  virtual void
    Set_TimeoutType( FSP_DirectiveTimeoutType type ) = 0;
  virtual void
    Set_AbortVC() = 0;
  virtual void
    Set_ModifyMapPriorityList( FSP_AbsolutePriority* plist,
                               int size ) = 0;
  virtual void
    Set_ModifyMapPollingVector( FSP_MapId* pvec, int size ) = 0;
  virtual void
    Set_InvokeDirectiveDiagnostic
   ( FSP_InvokeDirectiveDiagnostic diag ) = 0;
};
```

Methods

FSP_DirectiveId Get_DirectiveId() const;

Returns the directive identification.

FSP_DirectiveId Get_ExpectedDirectiveId() const;

Returns the next directive identification expected by the provider.

FSP_Directive Get_Directive() const;

Returns the directive.

unsigned long Get_VR() const;

Returns the requested value of the receiver frame sequence number V(R).

Precondition: the directive is ‘initiate AD with set V(R)’.

unsigned long Get_VS() const;

Returns the requested value of the transmitter frame sequence number V(S).

Precondition: the directive is ‘set V(S)’.

```
unsigned long Get_FopSlidingWindowWidth() const;
```

Returns the requested width of the FOP sliding window.

Precondition: the directive is 'set FOP sliding window width'.

```
unsigned long GetTimerInitial() const;
```

Returns the requested timeout value in microseconds.

Precondition: the directive is 'set T1 initial'.

```
unsigned long Get_TransmissionLimit() const;
```

Returns the requested transmission limit.

Precondition: the directive is 'set transmission limit'.

```
FSP_DirectiveTimeoutType Get_TimeoutType() const;
```

Returns the requested timeout type.

Precondition: the directive is 'set timeout type'.

```
FSP_AbsolutePriority* Get_Priority( int& size ) const;
```

Returns an array with the requested priorities for the MAPs if the list was set in the object (the multiplexing scheme is 'absolute priority'). Otherwise returns a NULL pointer.

Precondition: the directive is 'modify MAP multiplexing control'.

```
FSP_MapId* Get_PollingVector( int& size ) const;
```

Returns the requested MAP polling vector if the vector was set in the object (the multiplexing scheme is 'polling vector'). Otherwise returns a NULL pointer.

Precondition: the directive is 'modify MAP multiplexing control'.

```
FSP_InvokeDirectiveDiagnostic Get_InvokeDirectiveDiagnostic() const;
```

Returns the diagnostics code.

Precondition: the result is negative, and the diagnostic type is set to 'specific'.

```
void Set_DirectiveId( FSP_DirectiveId id );
```

Sets the parameter 'directive identification' to the value passed as argument.

void Set_ExpectedDirectiveId(FSP_DirectiveId id);

Sets the parameter 'expected directive identification' to the value passed as argument.

void Set_InitiateADwithoutCLCW();

Sets the parameter 'directive' to 'initiate AD without CLCW'.

void Set_InitiateADwithCLCW();

Sets the parameter 'directive' to 'initiate AD with CLCW'.

void Set_InitiateADwithUnlock();

Sets the parameter 'directive' to 'initiate AD with Unlock'.

void Set_InitiateADwithSetVR(unsigned long vr);

Sets the parameter 'directive' to 'initiate AD with Set V(R)' and stores the requested value of the receiver frame sequence number V(R).

Arguments

`vr` the requested value of V(R) in the range 0 to 255

void Set_TerminateAD();

Sets the parameter 'directive' to 'terminate AD'.

void Set_ResumeAD();

Sets the parameter 'directive' to 'resume AD'.

void Set_VS(unsigned long vs);

Sets the parameter 'directive' to 'set V(S)' and stores the requested value of the transmitter frame sequence number V(S).

Arguments

`vs` the requested value of V(S) in the range 0 to 255

void Set_FopSlidingWindow(unsigned long width);

Sets the parameter 'directive' to 'set FOP sliding window width' and stores the requested value of the window width.

<u>Arguments</u>

`width` the requested window width in the range 1 to 255

`void Set_TimerInitial( unsigned long timeout );`

Sets the parameter 'directive' to 'set T1 initial' and stores the requested value of the timeout.

<u>Arguments</u>

`timeout` the requested timeout value in microseconds

`void Set_TransmissionLimit( unsigned long limit );`

Sets the parameter 'directive' to 'set transmission limit' and stores the requested value of the limit.

<u>Arguments</u>

`limit` the requested transmission limit in the range 1 to 255

`void Set_TimeoutType( FSP_DirectiveTimeoutType type );`

Sets the parameter 'directive' to 'set timeout type' and stores the requested value passed as argument.

<u>Arguments</u>

`type` the requested timeout type

`void Set_AbortVC();`

Sets the parameter 'directive' to 'abort VC'.

```
void
Set_ModifyMapPriorityList(FSP_AbsolutePriority* plist, int size);
```

Sets the parameter 'directive' to 'modify MAP multiplexing control' and stores the priority list passed as argument. This method must be used if the multiplexing scheme is 'absolute priority'. Clears the polling vector if it is set.

<u>Arguments</u>

`plist` the requested priority list as an array of 1 to 64 MAP ID / Priority pairs

`size` the number of elements in the array (1 - 64)

```
void Set_ModifyMapPollingVector( FSP_MapId* pvec, int size );
```

Sets the parameter 'directive' to 'modify MAP multiplexing control' and stores the polling vector passed as argument. This method must be used if the multiplexing scheme is 'polling vector'. Clears the priority list if it is set.

Arguments

`pvec` the requested polling vector as an array of 1 to 192 MAP IDs

`size` the number of elements in the array (1 - 192)

```
void
Set_InvokeDirectiveDiagnostic(FSP_InvokeDirectiveDiagnostic diag);
```

Sets the result to 'negative', the diagnostic type to 'specific', and stores the value of the diagnostic code passed by the argument.

Initial Values of Operation Parameters after Creation

Parameter	Created directly	Created by Service Instance
directive identification	0	0
directive	'invalid'	'invalid'
V(R)	0	0
V(S)	0	0
FOP sliding window width	0	0
T1 initial	0	0
transmission-limit	0	0
timeout-type	'invalid'	'invalid'
map-multiplexing-control (priority list)	NULL	NULL
map-multiplexing-control (polling vector)	NULL	NULL
INVOKE DIRECTIVE diagnostic	'invalid'	'invalid'

Checking of Invocation Parameters

The interface ensures consistency between the directive and the directive parameters, as the client cannot set the directive directly. The consistency checks defined below only need to be performed when the invocation is received by the service provider. The method `VerifyInvocationArguments()` should nevertheless be called on the user side to check the permissible range of parameter arguments.

Parameter	Required condition
directive	Must not be 'invalid'
V(R)	If the directive is 'initiate AD with set V(R)' must be a value in the range 0 to 255.
V(S)	If the directive is 'set V(S)' must be a value in the range 0 to 255.
FOP sliding window width	If the directive is 'set FOP sliding window width' must be a value in the range 1 to 255
T1 initial	If the directive is 'set T1 initial' must not be zero.
transmission-limit	If the directive is 'set transmission limit' must not be zero
timeout-type	If the directive is 'set timeout type' must not be 'invalid'
map-multiplexing-control (priority list)	If the directive is 'modify map multiplexing control' must one of the following an array of 1 to 64 MAP ID / Priority pairs where each MAP ID must be in the range 0 to 63 and each priority in the range 1 to 64; NULL (multiplexing scheme is 'FIFO' or 'polling vector')
map-multiplexing-control (polling vector)	If the directive is 'modify map multiplexing control' must be one of the following an array of 1 to 192 MAP Ids where each MAP ID must be in the range 0 to 63; NULL (multiplexing scheme is 'FIFO' or 'absolute priority').

Checking of Return Parameters

Parameter	Required condition
invoke directive diagnostic	must not be 'invalid' if the result is 'negative' and the diagnostic type is 'specific'
expected directive id	If result is 'positive', must be directive id + 1

A4 FSP SERVICE INSTANCE INTERFACES

A4.1 SERVICE INSTANCE CONFIGURATION

Name `IFSP_SIAdmin`
GUID `{91DCEBA6-E896-11d4-9F17-00104B4F22C0}`
Inheritance: `IUnknown`
File `IFSP_SIAdmin.H`

The interface provides write and read access to the FSP-specific service instance configuration parameters. All configuration parameters must be set as part of service instance configuration. When the method `ConfigCompleted()` is called on the interface `ISLE_SIAdmin`, the service element checks that all parameters have been set and returns an error when the configuration is not complete.

FSP-specific configuration parameters are not processed or modified by the API. They are only used for the following purposes:

– to inform the service user via the GET-PARAMETER operation;

– to initialize parameters of the status report; or

– to check operation parameters.

FSP configuration parameters can be modified at any time. The API always uses the last value set in GET-PARAMETER returns. Parameters used for initialization of the status report must not be set after invocation of `ConfigCompleted()`. The effect of invoking these methods at a later stage is undefined.

As a convenience for the application, the interface provides read access to the configuration parameters, except for parameters used to initialize the status report. If retrieval methods are called before configuration, the value returned is undefined.

It is noted that service management might constrain the range of parameters that can be modified after configuration. The API does not enforce these constraints.

In addition to the FSP configuration parameters accessible via this interface, the FOP parameters controlled via the interface `IFSP_FOPMonitor` must be initialized before calling `ConfigCompleted()`.

CCSDS HISTORICAL DOCUMENT

Synopsis

```
#include <FSP_Types.h>
#include <SLE_SCM.H>

#define IID_IFSP_SIAdmin_DEF { 0x91dceba6, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_SIAdmin : IUnknown
{
  virtual void
    Set_MaximumFrameLength( unsigned int length ) = 0;
  virtual void
    Set_MaximumPacketLength( unsigned int length ) = 0;
  virtual void
    Set_VcMuxScheme( FSP_MuxScheme scheme ) = 0;
  virtual void
    Set_VcPriorityList( const FSP_AbsolutePriority* priorities,
                        int size ) = 0;
  virtual void
    Set_VcPollingVector( const FSP_VcId* pvec, int size ) = 0;
  virtual void
    Set_BlockingTimeout( unsigned long timeout ) = 0;
  virtual void
    Set_BlockingUsage( FSP_BlockingUsage usage ) = 0;
  virtual void
    Set_DirectiveInvocationEnabled( SLE_YesNo yesNo ) = 0;
  virtual void
    Set_SegmentHeaderPresent( SLE_YesNo yesNo ) = 0;
  virtual void
    Set_ApIdList( const FSP_ApId* plist, int size ) = 0;
  virtual void
    Set_MapList( const FSP_MapId* plist, int size ) = 0;
  virtual void
    Set_VirtualChannel( FSP_VcId id ) = 0;
  virtual void
    Set_PermittedTransmissionMode( FSP_PermittedTransmissionMode mode)= 0;
  virtual void
    Set_MaximumBufferSize( FSP_BufferSize size ) = 0;
  virtual void
    Set_InitialProductionStatus( FSP_ProductionStatus status ) = 0;
  virtual void
    Set_InitialDirectiveInvocationOnline( SLE_YesNo yesNo ) = 0;
  virtual unsigned int
    Get_MaximumFrameLength() const = 0;
  virtual unsigned int
    Get_MaximumPacketLength() const = 0;
  virtual FSP_MuxScheme
    Get_VcMuxScheme() const = 0;
  virtual const FSP_AbsolutePriority*
    Get_VcPriorityList( int& size ) const = 0;
  virtual const FSP_VcId*
    Get_VcPollingVector( int& size ) const = 0;
  virtual unsigned long
    Get_BlockingTimeout() const = 0;
  virtual FSP_BlockingUsage
    Get_BlockingUsage() const = 0;
  virtual SLE_YesNo
    Get_DirectiveInvocationEnabled() const = 0;
```


CCSDS HISTORICAL DOCUMENT

```
  virtual SLE_YesNo
    Get_SegmentHeaderPresent() const = 0;
  virtual const FSP_ApId*
    Get_ApIdList( int& size ) const = 0;
  virtual const FSP_MapId*
    Get_MapList( int& size ) const = 0;
  virtual FSP_VcId
    Get_VirtualChannel() const = 0;
  virtual FSP_PermittedTransmissionMode
    Get_PermittedTransmissionMode() const = 0;
  virtual FSP_BufferSize
    Get_MaximumBufferSize() const = 0;
};
```

Methods

`void Set_MaximumFrameLength( unsigned int length );`

Sets the mission maximum TC transfer frame length in octets.

Arguments

`length` a number in the range 12 to 1026 octets

`void Set_MaximumPacketLength( unsigned int length );`

Sets the mission maximum telecommand packet length in octets.

Arguments

`length` a number in the range 7 to 65542 octets

`void Set_VcMuxScheme( FSP_MuxScheme scheme );`

Sets the VC multiplexing scheme in effect: (‘FIFO’, ‘absolute priority’, ‘polling vector’).

```
void
Set_VcPriorityList( const FSP_AbsolutePriority* priorities,
                    int size );
```

Sets the priority list for the VC multiplexing scheme ‘absolute priority’. Must not be set if the VC multiplexing scheme is ‘FIFO’ or ‘polling vector’.

Arguments

`priorities` an array of VC ID priority pairs as defined by the type `FSP_AbsolutePriority`

`size` the number of elements in the list

void Set_VcPollingVector(const FSP_VcId* pvec, int size);

Sets the polling vector for the VC multiplexing scheme 'polling vector'. Must not be set if the VC multiplexing scheme is 'FIFO' or 'absolute priority'.

Arguments

`pvec` an array of VC IDs in the sequence the VCs are polled

`size` the number of elements in the vector

void Set_BlockingTimeout(unsigned long timeout);

Sets the period from inserting the first packet into the TC frame data unit until this unit is passed to the FOP regardless of the number of packets contained. Shall not be set when blocking usage is set to 'permitted'.

Arguments

`timeout` timeout period in microseconds

void Set_BlockingUsage(FSP_BlockingUsage usage);

Defines whether packet blocking is permitted on the VC.

void Set_DirectiveInvocationEnabled(SLE_YesNo yesNo);

Defines whether the service instance being configured is allowed to invoke directives. The argument shall be set to 'yes' if this service instance is authorized to invoke the FSP-INVOKE-DIRECTIVE operation, and to 'no' otherwise.

void Set_SegmentHeaderPresent(SLE_YesNo yesNo);

Specifies whether a segment header is present ('yes') or absent ('no') in the TC transfer frames.

void Set_ApIdList(const FSP_ApId* plist, int size);

Specifies the list of APIDs the given service instance is authorized to access.

Arguments

`plist` array of APIDs, each APID is in the range 0 to 2047

`size` number of APIDs in the array (1 to 2048)

void Set_MapList(const FSP_MapId* plist, int size);

Specifies the list of MAPs permitted to be used by the given service instance if MAPs are used. Must not be set when MAPs are not used.

void Set_VirtualChannel(FSP_VcId id);

Specifies the virtual channel used by this service instance.

void
Set_PermittedTransmissionMode(FSP_PermittedTransmissionMode mode);

Specifies the transmission mode permitted to be used by the given service instance.

void Set_MaximumBufferSize(FSP_BufferSize size);

Specifies the maximum packet buffer size in units of octets. This value is used to initialize the status parameter packet buffer available.

Precondition: `ISLE_SIAdmin::ConfigCompleted()` was not called yet.

void Set_InitialProductionStatus(FSP_ProductionStatus status);

Sets the production status at the time of service instance configuration. The value is used to initialize the status parameter production status. The current value of the production status can be retrieved via the interface `IFSP_SIUpdate`.

Precondition: `ISLE_SIAdmin::ConfigCompleted()` was not called yet.

void Set_InitialDirectiveInvocationOnline(SLE_YesNo yesNo);

Specifies whether a service instance with directive invocation capability is connected at the time of configuration. This method only needs to be called when directive invocation is not enabled for the service instance. If directive invocation is enabled, the method invocation is ignored. The current value of the parameter can be retrieved via the interface `IFSP_SIUpdate`.

Precondition: `ISLE_SIAdmin::ConfigCompleted()` was not called yet.

unsigned int Get_MaximumFrameLength() const;

Returns the mission maximum TC transfer frame length in octets.

unsigned int Get_MaximumPacketLength() const;

Returns the maximum packet length.

FSP_MuxScheme Get_VcMuxScheme() const;

Returns the VC multiplexing scheme in effect.

const FSP_AbsolutePriority* Get_VcPriorityList(int& size) const;

Returns the priority list as a vector of VC ID/priority pairs if the VC multiplexing scheme is 'absolute priority'. If the multiplexing scheme is 'FIFO' or 'polling vector', returns NULL.

const FSP_VcId* Get_VcPollingVector(int& size) const;

Returns the polling vector as an array of VCIDs if the VC multiplexing scheme is 'polling vector'. If the multiplexing scheme is 'FIFO' or 'absolute priority', returns NULL.

unsigned long Get_BlockingTimeout() const;

Returns the blocking timeout period.

Precondition: `Get_BlockingUsage()` returns 'permitted'.

FSP_BlockingUsage Get_BlockingUsage() const;

Returns whether blocking of packets is permitted.

SLE_YesNo Get_DirectiveInvocationEnabled() const;

Returns 'yes' if the service instance is allowed to invoke the FSP-INVOKE-DIRECTIVE operation and 'no' otherwise.

SLE_YesNo Get_SegmentHeaderPresent() const;

Returns 'yes' if a segment header is present in the TC transfer frames, and 'no' otherwise.

const FSP_ApId* Get_ApIdList(int& size) const;

Returns the list of APIDs the service instance is authorized to access.

const FSP_MapId* Get_MapList(int& size) const;

Returns the list of MAPs the service instance is authorized to access.

FSP_VcId Get_VirtualChannel() const;

Returns the virtual channel used by the service instance.

```
FSP_PermittedTransmissionMode Get_PermittedTransmissionMode() const;
```

Returns the transmission mode the service instance is authorized to use.

```
FSP_BufferSize Get_MaximumBufferSize() const;
```

Returns the maximum packet buffer size.

A4.2 FOP MONITORING AND CONTROL

Name `IFSP_FOPMonitor`
GUID `{D9E3A601-641A-11d5-9CF0-0004761E8CFB}`
Inheritance: `IUnknown`
File `IFSP_FOPMonitor.H`

The interface provides access to the FSP parameters related to the FOP machine of the VC on which the service instance operates including

- parameters controlling operation the FOP machine; and
- parameters monitoring the FOP state and variables.

The API service instance uses these parameters only to respond to GET-PARAMETER invocations. All parameters must be set when the service instance is being configured before the method `ConfigCompleted()` is called on the interface `ISLE_SIAdmin`. The service instance verifies completeness and consistency of the parameters within the method `ConfigCompleted()`.

During the lifetime of the service instance, FOP related parameters must be updated whenever they change. Changes might occur because of directives invoked by a service user on the same or on a different service instance, because of events detected by the FOP machine, or because of management action. In order to ensure that the service instance always reports the correct parameter value, updates must be reported independent of the service instance state.

The parameters 'map-multiplexing-scheme' and 'map-multiplexing-control' are included in this interface because 'map-multiplexing-control' can be modified by the service user via a directive.

Synopsis

```
#include <FSP_Types.h>
#include <SLE_SCM.H>

#define IID_IFSP_FOP_DEF {0xd9e3a601, 0x641a, 0x11d5, \
           { 0x9c, 0xf0, 0x0, 0x4, 0x76, 0x1e, 0x8c, 0xfb} }

interface IFSP_FOPMonitor : IUnknown
{
  virtual void
    Set_FopSlidingWindow( unsigned long window ) = 0;
  virtual void
    Set_TimeoutType( FSP_TimeoutType type ) = 0;
  virtual void
    Set_TimerInitial( unsigned long timeout ) = 0;
  virtual void
    Set_TransmissionLimit( unsigned long limit ) = 0;
  virtual void
    Set_TransmitterFrameSequenceNumber( unsigned long number ) = 0;
```

```
  virtual void
    Set_FopState( FSP_FopState state ) = 0;
  virtual void
    Set_MapPriorityList( const FSP_AbsolutePriority* priorities,
                         int size ) = 0;
  virtual void
    Set_MapPollingVector( const FSP_MapId* pvec, int size ) = 0;
  virtual void
    Set_MapMuxScheme( FSP_MuxScheme scheme ) = 0;
  virtual unsigned long
    Get_FopSlidingWindow() const = 0;
  virtual FSP_TimeoutType
    Get_TimeoutType() const = 0;
  virtual unsigned long
    Get_TimerInitial() const = 0;
  virtual unsigned long
    Get_TransmissionLimit() const = 0;
  virtual unsigned long
    Get_TransmitterFrameSequenceNumber() const = 0;
  virtual FSP_FopState
    Get_FopState() const = 0;
  virtual const FSP_AbsolutePriority*
    Get_MapPriorityList( int& size ) const = 0;
  virtual const FSP_MapId*
    Get_MapPollingVector( int& size ) const = 0;
  virtual FSP_MuxScheme
    Get_MapMuxScheme() const = 0;
};
```

Methods

`void Set_FopSlidingWindow( unsigned long window );`

Sets the FOP sliding window width, i.e., the number of frames that can be transmitted on the given VC before an acknowledgement is required.

<u>Arguments</u>

`window` a number in the range 1 to 255

`void Set_TimeoutType( FSP_TimeoutType type );`

Specifies the FOP behavior in case of a timeout (‘Alert’ or ‘AD service suspension’).

`void Set_TimerInitial( unsigned long timeout );`

Specifies the initial value for countdown timer when an AD or BC frame is transmitted.

<u>Arguments</u>

`timeout` timer value in microseconds

```
void Set_TransmissionLimit( unsigned long limit );
```

Specifies the maximum number of times the first frame on the Sent Queue may be transmitted.

```
void Set_TransmitterFrameSequenceNumber( unsigned long number );
```

Sets the current value of the FOP Transmitter Frame Sequence Number, V(S), which contains the value of the Frame Sequence Number, N(S), to be put in the Transfer Frame Header of the next Type AD frame to be transmitted. The parameter shall be updated, whenever the transmission mode capability changes (i.e., when the sequence controlled service is suspended, terminated, or started).

Arguments

`number` V(S) value in the range 0 to 255

```
void Set_FopState( FSP_FopState state );
```

Sets the current value of the FOP state. The parameter shall be updated for every changed of the FOP state.

```
void
Set_MapPriorityList( const FSP_AbsolutePriority* priorities,
                     int size );
```

Sets the priority list for the MAP multiplexing scheme 'absolute priority'. Must not be set if the MAP multiplexing scheme is 'FIFO' or 'polling vector'.

Arguments

`priorities` an array of MAP ID priority pairs as defined by the type `FSP_AbsolutePriority`

`size` the number of elements in the list

```
void Set_MapPollingVector( const FSP_MapId* pvec, int size );
```

Sets the polling vector for the MAP multiplexing scheme 'polling vector'. Must not be set if the MAP multiplexing scheme is 'FIFO' or 'absolute priority'.

Arguments

`pvec` an array of MAP IDs in the sequence the MAPs are polled

`size` the number of elements in the vector

void Set_MapMuxScheme(FSP_MuxScheme scheme);

Sets the MAP multiplexing scheme in effect: (‘FIFO’, ‘absolute priority’, ‘polling vector’).

unsigned long Get_FopSlidingWindow() const;

Returns the FOP sliding window width.

FSP_TimeoutType Get_TimeoutType() const;

Returns the current setting of the timeout type (‘Alert’ or ‘AD service suspension’).

unsigned long Get_TimerInitial() const;

Returns the initial value of the FOP countdown timer in microseconds.

unsigned long Get_TransmissionLimit() const;

Returns the FOP (re-)transmission limit.

unsigned long Get_TransmitterFrameSequenceNumber() const;

Returns the current value of the FOP variable V(S)—this value is only updated when the transmission mode capability changes.

FSP_FopState Get_FopState() const;

Returns the current value of the FOP state.

const FSP_AbsolutePriority* Get_MapPriorityList(int& size) const;

Returns the priority list as a vector of MAP ID/priority pairs if the MAP multiplexing scheme is ‘absolute priority’. If the multiplexing scheme is ‘FIFO’ or ‘polling vector’, returns NULL.

const FSP_MapId* Get_MapPollingVector(int& size) const;

Returns the polling vector as an array of MAP IDs if the MAP multiplexing scheme is ‘polling vector’. If the multiplexing scheme is ‘FIFO’ or ‘absolute priority’, returns NULL.

FSP_MuxScheme Get_MapMuxScheme() const;

Returns the MAP multiplexing scheme in effect.

A4.3 UPDATE OF SERVICE INSTANCE PARAMETERS

Name `IFSP_SIUpdate`
GUID `{91DCEBA7-E896-11d4-9F17-00104B4F22C0}`
Inheritance: `IUnknown`
File `IFSP_SIUpdate.H`

The interface provides methods to update parameters that shall be reported to the service user via the operations FSP-STATUS-REPORT, FSP-ASYNC-NOTIFY, and FSP-GET-PARAMETER. In order to keep this information up to date the appropriate methods of this interface must be called whenever certain events occur (see the specification in 3.1). If these events must be reported to the FSP service user via a notification, the API can be requested to send the notification. Alternatively the application can generate and send the notification itself.

The methods of this interface must always be called when one of the relevant events occurs, independent of the state of the service instance. Notifications to the user will only be sent, if the service instance state is either ‘ready’ or ‘active’. Failure to inform the API of an event can result in incorrect and inconsistent parameters in the status report.

Because of performance considerations, methods processing nominal events perform no plausibility checks, but completely rely on the application to provide correct and consistent arguments.

The interface provides read access to the parameters set via this interface and to parameters accumulated or derived by the API according to the specifications in 3.1.4. The API sets the parameters to the initial values specified at the end of this annex when the service instance is configured. Parameter values retrieved before configuration are undefined.

Synopsis

```
#include <FSP_Types.h>
#include <SLE_SCM.H>
interface ISLE_Time;

#define IID_IFSP_SIUpdate_DEF { 0x91dceba7, 0xe896, 0x11d4, \
          { 0x9f, 0x17, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface IFSP_SIUpdate : IUnknown
{
  virtual void
    PacketStarted( FSP_PacketId packetId,
                   FSP_TransmissionMode mode,
                   const ISLE_Time& startTime,
                   FSP_BufferSize bufferAvailable
                   bool notify ) = 0;
  virtual void
    PacketRadiated( FSP_PacketId packetId,
                    FSP_TransmissionMode mode,
                    const ISLE_Time& radiationTime,
```

```
                  bool notify ) = 0;
virtual void
  PacketAcknowledged( FSP_PacketId packetId,
                      const ISLE_Time& ackTime,
                      bool notify ) = 0;
virtual void
  BufferEmpty( bool notify ) = 0;
virtual HRESULT
  PacketNotStarted( FSP_PacketId packetId,
                    FSP_TransmissionMode mode,
                    const ISLE_Time& startTime,
                    FSP_Failure reason,
                    const FSP_PacketId* affectedPackets,
                    int numAffected,
                    FSP_BufferSize bufferAvailable,
                    bool notify ) = 0;
virtual HRESULT
  ProductionStatusChange( FSP_ProductionStatus newStatus,
                          const FSP_PacketId* affectedPackets,
                          int numAffected,
                          FSP_FopAlert fopAlert,
                          FSP_BufferSize bufferAvailable,
                          bool notify ) = 0;
virtual HRESULT
  VCAborted( const FSP_PacketId* affectedPackets,
             int numAffected,
             FSP_BufferSize bufferAvailable,
             bool notify ) = 0;
virtual HRESULT
  NoDirectiveCapability( bool notify ) = 0;
virtual HRESULT
  DirectiveCapabilityOnline( bool notify ) = 0;
virtual HRESULT
  DirectiveCompleted( FSP_DirectiveId directiveId,
                      SLE_Result result,
                      FSP_FopAlert fopAlert,
                      bool notify ) = 0;
virtual HRESULT
  EventProcCompleted( FSP_EventInvocationId eventId,
                      FSP_EventResult result,
                      bool notify ) = 0;
virtual FSP_ProductionStatus
  Get_ProductionStatus() const = 0;
virtual SLE_YesNo
  Get_DirectiveInvocationOnline() const = 0;
virtual FSP_BufferSize
  Get_PacketBufferAvailable() const = 0;
virtual unsigned long
  Get_NumberOfADPacketsReceived() const = 0;
virtual unsigned long
  Get_NumberOfBDPacketsReceived() const = 0;
virtual unsigned long
  Get_NumberOfADPacketsProcessed() const = 0;
virtual unsigned long
  Get_NumberOfBDPacketsProcessed() const = 0;
virtual unsigned long
  Get_NumberOfADPacketsRadiated() const = 0;
virtual unsigned long
  Get_NumberOfBDPacketsRadiated() const = 0;
virtual unsigned long
```

```
    Get_NumberOfPacketsAcknowledged() const = 0;
  virtual FSP_PacketId
    Get_PacketLastProcessed() const = 0;
  virtual const ISLE_Time*
    Get_ProductionStartTime() const = 0;
  virtual FSP_PacketStatus
    Get_PacketStatus() const = 0;
  virtual FSP_PacketId
    Get_PacketLastOk() const = 0;
  virtual const ISLE_Time*
    Get_ProductionStopTime() const = 0;
  virtual FSP_PacketId
    Get_ExpectedPacketId() const = 0;
  virtual FSP_DirectiveId
    Get_ExpectedDirectiveInvocationId() const = 0;
  virtual FSP_EventInvocationId
    Get_ExpectedEventInvocationId() const = 0;
};
```

Methods

```
void
PacketStarted( FSP_PacketId packetId,
               FSP_TransmissionMode mode,
               const ISLE_Time& startTime,
               FSP_BufferSize bufferAvailable
               bool notify );
```

The method shall be called when processing of a packet has been started, i.e., the packet has been extracted from the packets queue and forwarded to the segments queue. It performs the following actions:

a) if the value of mode is ‘sequence controlled’, increments the number of AD packets processed otherwise increments the number of BD packets processed;

b) stores the value of the argument `packetId` to the parameter `packet-identification-last-processed`;

c) copies the `startTime` to the parameter `production-start-time`;

d) sets the parameter `packet-status` to ‘production started’;

e) sets the parameter `packet-buffer-available` to the value passed by the argument `bufferAvailable`;

f) if the argument notify is true:

   1) creates an empty packet identification list and inserts the argument `packetId` into that parameter;

   2) sends the notification ‘packet processing started’ to the service user provided sending of notifications is allowed according to the state tables in reference [5].

Preconditions

The client must ensure the following preconditions since they are not checked by the implementation:

a) the production status must be 'operational';

b) if the transmission mode is 'sequence controlled' the production status must be 'operational AD and BD';

c) the argument notify must only be set to TRUE if the service user has requested a 'processing started' notification for the packet.

Arguments

`packetId`	the identification of the packet for which processing started
`mode`	the transmission mode ('sequence controlled' or 'expedited') of the packet
`startTime`	the time at which processing of the packet started
`bufferAvailable`	the amount of packet buffer currently available
`notify`	if true a notification shall be sent to the service user

```
void
PacketRadiated( FSP_PacketId packetId,
                FSP_TransmissionMode mode,
                const ISLE_Time& radiationTime,
                bool notify );
```

The method shall be called when a packet has been completely radiated. If segmentation is used, this implies that the last segment of the packet was radiated. The method performs the following actions:

a) if the value of mode is 'sequence controlled', increments the number of AD packets radiated otherwise increments the number of BD packets radiated;

b) if the value of mode is 'expedited', sets the parameter packet identification last OK to the value of the argument `packetId` and copies the `raditationTime` to the parameter `production-stop-time`;

c) if the argument `packetId` equals the parameter `packet-identification-last-processed`, sets the parameter packet status to 'packet radiated';

d) if the argument notify is true:

   1) creates an empty packet identification list and inserts the argument `packetId` into that parameter;

2) sends the notification ‘packet radiated’ to the service user provided sending of notifications is allowed according to the state tables in reference [5].

Preconditions

The client must ensure the following preconditions since they are not checked by the implementation:

a) the production status must be ‘operational’;

b) if the transmission mode is ‘sequence controlled’ the production status must be ‘operational AD and BD’;

c) the argument notify must only be set to TRUE if the service user has requested a ‘radiated’ notification for the packet.

Arguments

`packetId`	the identification of the packet that was radiated
`mode`	the transmission mode (‘sequence controlled’ or ‘expedited’) of the packet
`radiationTime`	the time at which radiation completed
`notify`	if true a notification shall be sent to the service user

```
void
PacketAcknowledged( FSP_PacketId packetId,
                    const ISLE_Time& ackTime,
                    bool notify );
```

The method shall be called when all components of a packet were acknowledged by the space element via the associated stream of CLCWs. It performs the following actions:

a) increments the parameter number of packets acknowledged;

b) sets the parameter packet identification last OK to the value of the argument `packetId`;

c) copies `ackTime` to the parameter ‘production stop time’;

d) if the argument `packetId` equals the parameter packet identification last processed, sets the parameter packet status to ‘packet acknowledged’;

e) if the argument notify is true:

1) creates an empty packet identification list and inserts the argument `packetId` into that parameter;

CCSDS HISTORICAL DOCUMENT

2) sends the notification 'packet acknowledged' to the service user provided sending of notifications is allowed according to the state tables in reference [5].

Preconditions

The argument notify must only be set to TRUE if the service user has requested an 'acknowledged' notification for the packet.

Arguments

`packetId` the identification of the packet that was acknowledged

`ackTime` the time at which the last component of the packet was acknowledged

`notify` if true a notification shall be sent to the service user

```
void BufferEmpty( bool notify );
```

The method shall be called when the packet buffer becomes empty because all packets were processed. It shall not be called when the packet buffer is cleared because of one of the events for which reference [3] requires discarding of buffered packets.

The method performs the following actions:

a) sets the parameter packet buffer available to the maximum buffer size set by configuration of the service instance;

b) if the argument notify is true, sends the notification 'buffer empty' to the service user provided sending of notifications is allowed according to the state tables in reference [5].

Arguments

`notify` if true a notification shall be sent to the service user

```
HRESULT
PacketNotStarted( FSP_PacketId packetId,
                  FSP_TransmissionMode mode,
                  const ISLE_Time& startTime,
                  FSP_Failure reason,
                  const FSP_PacketId* affectedPackets,
                  int numAffected,
                  FSP_BufferSize bufferAvailable,
                  bool notify );
```

The method shall be called when processing of a packet could not be started because:

– the latest production start time was expired;

– the production status was 'interrupted'; or

CCSDS HISTORICAL DOCUMENT

CCSDS HISTORICAL DOCUMENT

– the required transmission mode was not available.

It performs the following actions:

a) if the value of `mode` is 'sequence controlled', increments the number of AD packets processed otherwise increments the number of BD packets processed;

b) sets the parameter packet identification last processed to the value of the argument `packetId`;

c) copies the `startTime` to the parameter production start time;

d) sets the parameter packet status to 'expired', 'interrupted', or 'unsupported transmission mode' according to the value of the argument `reason`;

e) sets the parameter packet buffer available to the value passed by the argument `bufferAvailable`;

f) if the argument `notify` is true and sending of notifications is allowed according to the state tables in reference [5]:

   1) creates an empty packet identification list and inserts the argument `packetId` and all identifiers in the argument `affectedPackets` into that parameter;

   2) if `reason` is 'expired', sends the notification 'sldu expired' to the service user;

   3) if `reason` is 'interrupted', sends the notification 'production interrupted' to the service user;

   4) if `reason` is 'transmission mode mismatch', sends the notification 'transmission mode mismatch' to the service user.

Arguments

`packetId`	the identification of the packet for which processing could not be started
`mode`	the transmission mode ('sequence controlled' or 'expedited') of the packet
`startTime`	the time at which processing of the packet was attempted
`reason`	the reason why processing could not be started
`affectedPackets`	an array containing the identifiers of all packets (excluding the identifier passed by `packetId`) that were or will be discarded because of the problem detected. If `packetId` is the only affected packet, a NULL pointer shall be supplied and the argument `numAffected` shall be set to zero.
`numAffected`	the number of packet identifiers in the array `affectedPackets`

CCSDS HISTORICAL DOCUMENT

`bufferAvailable` the amount of packet buffer currently available

`notify` if true a notification shall be sent to the service user

Result codes

`S_OK` the updates have been made and the notification was sent if requested

`SLE_E_INCONSISTENT` the arguments supplied are inconsistent (see NOTE)—updates have not been performed and a notification has not been sent

`SLE_E_STATE` the service instance state is 'unbound' (it might have aborted)—updates have been performed but the requested notification could not be sent

NOTE – The result code `SLE_E_INCONSISTENT` indicates one of the following problems:

a) `reason` is 'interrupted' but the production status is not 'interrupted' or 'halted'; or

b) `reason` is 'transmission mode mismatch' but `mode` is 'expedited'.

```
HRESULT
ProductionStatusChange( FSP_ProductionStatus newStatus,
                        const FSP_PacketId* affectedPackets,
                        int numAffected,
                        FSP_FopAlert fopAlert,
                        FSP_BufferSize bufferAvailable,
                        bool notify );
```

The method shall be called whenever the production status changes, including changes of the operational sub-states. It performs the following actions:

a) sets the parameter production status to `newStatus`;

b) if the argument `affectedPackets` is not NULL and contains the identifier stored in the parameter packet identification last processed, sets the parameter packet status to

– 'interrupted' if the new production status is 'halted' or 'interrupted', or

– 'unsupported transmission mode' if the new production status is 'operational';

c) sets the parameter packet buffer available to the value passed by the argument bufferAvailable;

d) if the argument notify is true and sending of notifications is allowed according to the state tables in reference [5]:

1) if the argument `affectedPackets` is not NULL and not empty, creates an empty packet identification list and copies all identifiers in the argument `affectedPackets` to that parameter;

2) if the production status changed to ‘halted’ sends the notification ‘production halted’ to the service user;

3) if the production status changed to ‘interrupted’ and the argument `affectedPackets` is not NULL and not empty, sends the notification ‘production interrupted’ to the service user;

4) if the production status changed from ‘operational AD and BD’ to ‘interrupted’ and the argument `affectedPackets` is NULL or empty, sends the notification ‘transmission mode capability change’ to the service user;

5) if the new production status is operational and a different production status value was previously reported to the user or no status was reported at all, sends the notification ‘production operational’ to the service user;

6) if the new production status is ‘operational BD’ or ‘operational AD suspended’ and the previous value of the production status was ‘operational AD and BD’ or if the new production status is ‘operational AD and BD’ and the previous value of the production status was ‘operational BD’ or ‘operational AD suspended’, sends the notification ‘transmission mode capability change’ to the service user;

7) if the new production status is ‘operational BD’ or ‘operational AD suspended’ and the argument `affectedPackets` is not NULL, sends the notification ‘packet transmission mode mismatch’ in addition to the notification ‘transmission mode capability change’.

<u>Arguments</u>

`newStatus`	the new value of the production status
`affectedPackets`	an array containing the identifiers of all packets that was or will be discarded because of the change of the production status. If no packets are affected by the change of the production status, a NULL pointer shall be supplied and the argument `numAffected` shall be set to zero.
`numAffected`	the number of packet identifiers in the array `affectedPackets`
`fopAlert`	the FOP Alert that caused the transmission mode capability change, if applicable. If the transmission mode capability did not change, the API ignores this argument.
`bufferAvailable`	the amount of packet buffer currently available

CCSDS HISTORICAL DOCUMENT

`notify` if true a notification shall be sent to the service user

Result codes

`S_OK`	the updates were made and a notification was sent if required and requested
`SLE_S_IGNORED`	the production status has not changed; the request was ignored
`SLE_E_INCONSISTENT`	the arguments supplied are inconsistent (see NOTE)—updates have not been performed and a notification has not been sent
`SLE_E_SEQUENCE`	there is no valid transition between the old and the new production status—this is only a warning, updates were made and notifications were sent if requested
`SLE_E_STATE`	the service instance state is 'unbound' (it might have aborted)—updates have been performed but the requested notification could not be sent

NOTES

1 If the production status did not change or the value of the new production status is 'configured' no notification is sent.

2 Valid transitions of the production status are defined in reference [3].

3 The result code `SLE_E_INCONSISTENT` indicates one of the following problems:

a) The argument `affectedPackets` is not NULL although:

– the new production status is either 'configured' or 'operational AD and BD';

– the old production status was not 'operational'; or

– the production status changed from 'operational AD suspended' to 'operational BD'.

b) The new production status is 'interrupted' or 'halted' and the packet status is 'production started' but the argument `affectedPackets` is NULL or the packet identification last processed in not contained in the list.

c) The old production status was 'configured' or 'interrupted' and the new status is 'operational' but the sub-state is not 'BD'.

4 If the production status change was caused by the directive 'abort VC' the method `VCAborted()` must be called instead of `ProductionStatusChange()`.

CCSDS HISTORICAL DOCUMENT

```
HRESULT
VCAborted( const FSP_PacketId* affectedPackets,
           int numAffected,
           FSP_BufferSize bufferAvailable,
           bool notify );
```

The method shall be called following successful execution of the directive ‘abort VC’. It performs the following actions:

a) sets the parameter production status to ‘operational BD’;

b) if the identifier in the parameter packet identification last processed is included in the argument `affectedPackets` sets the parameter packet status to ‘interrupted’;

c) sets the parameter packet buffer available to the value passed by the argument `bufferAvailable`;

d) if the argument `notify` is true:

   1) if `affectedPackets` is not NULL and not empty creates an empty packet identification list and copies all identifiers in the argument `affectedPackets` to that parameter;

   2) Sends the notification ‘VC aborted’ to the service user provided sending of notifications is allowed according to the state tables in reference [5].

The directive ‘VC aborted’ will generally cause a change of the production status to ‘operational BD’. This change is handled by the method VCAborted. Therefore, the method ProductionStatusChange() must not be called ion this case.

<u>Arguments</u>

`affectedPackets`	an array containing the identifiers of all packets that were or will be discarded. If no packets are affected by the directive ‘abort VC’, a NULL pointer shall be supplied and the argument `numAffected` shall be set to zero.
`numAffected`	the number of packet identifiers in the array `affectedPackets`
`bufferAvailable`	the amount of packet buffer currently available
`notify`	if true a notification shall be sent to the service user

Result codes

`S_OK`	the updates were made and the notification was sent if requested
`SLE_E_INCONSISTENT`	the packet status is 'production started' but the argument `bufferAvailable` is NULL or the packet identification last processed is not contained in the list—updates have not been performed and a notification has not been sent
`SLE_E_SEQUENCE`	the value of the production status was not 'operational' when the method was called—updates have not been performed and a notification has not been sent
`SLE_E_STATE`	the service instance state is 'unbound' (it might have aborted)—updates have been performed but the requested notification could not be sent

`HRESULT NoDirectiveCapability( bool notify );`

The method is called when the service instance that has the directive invocation capability for the VC is unbound following an UNBIND operation, a PEER-ABORT operation, or a protocol abort event. It performs the following actions if directive invocation is not enabled for the service instance:

a) sets the parameter 'directive invocation online' to 'no';

b) if the argument notify is true sends the notification 'no invoke directive capability on this VC' to the service user provided sending of notifications is allowed according to the state tables in reference [5].

Arguments

`notify`	if true a notification shall be sent to the service user

Result codes

`S_OK`	the notification was sent if requested
`SLE_S_IGNORED`	directive invocation is enabled for this service instance; the request has been ignored
`SLE_E_STATE`	the service instance state is 'unbound' (it might have aborted); the requested notification could not be sent

```
HRESULT DirectiveCapabilityOnline( bool notify );
```

The method is called when a service instance that has the directive invocation capability for the VC has successfully bound to the service provider. It performs the following actions if directive invocation is not enabled for the service instance:

a) sets the parameter 'directive invocation online' to 'yes';

b) if the argument notify is true sends the notification 'invoke directive capability on this VC established' to the service user provided sending of notifications is allowed according to the state tables in reference [5].

<u>Arguments</u>

`notify`	if true a notification shall be sent to the service user

<u>Result codes</u>

`S_OK`	the notification was sent if requested
`SLE_S_IGNORED`	directive invocation is enabled for this service instance; the request has been ignored
`SLE_E_STATE`	the service instance state is 'unbound' (it might have aborted); the requested notification could not be sent

```
HRESULT
DirectiveCompleted( FSP_DirectiveId directiveId,
                    SLE_Result result,
                    FSP_FopAlert fopAlert,
                    bool notify );
```

The method should be called when execution of a directive was completed successfully or failed. If the value of the argument result is 'positive result' the method generates and transmits the notification 'positive confirm response to directive'. If the result is 'negative result', it generates and transmits the notification 'negative confirm response to directive'.

<u>Arguments</u>

`directiveId`	the directive identification as contained in the FSP-INVOKE-DIRECTIVE invocation
`result`	the result of directive execution ('positive result' or 'negative result')
`fopAlert`	in case of a negative result, the FOP alert indicating why the directive failed; if result is 'positive result', the method ignores this argument

`notify` if true a notification shall be sent to the service user; because sending the notification is the only action of the method this argument is not really needed—it is provided for consistency with other methods in this interface

Result codes

`S_OK` the notification was sent if requested

`SLE_E_STATE` the service instance state is 'unbound' (it might have aborted); the requested notification could not be sent

```
HRESULT
EventProcCompleted( FSP_EventInvocationId eventId,
                    FSP_EventResult result,
                    bool notify );
```

The method should be called when processing of an event requested by an accepted FSP-THROW-EVENT operation completed. Depending on the value of the argument result, the method generates and transmits one of the notifications 'action list completed', 'action list not completed', or 'event condition evaluated to false'.

Arguments

`eventId` the event invocation identifier as contained in the FSP-THROW-EVENT invocation

`result` the result of event processing ('completed', 'not completed', or 'condition false')

`notify` if true a notification shall be sent to the service user; because sending the notification is the only action of the method this argument is not really needed—it is provided for consistency with other methods in this interface

Result codes

`S_OK` the notification was sent if requested

`SLE_E_STATE` the service instance state is 'unbound' (it might have aborted); the requested notification could not be sent

```
FSP_ProductionStatus Get_ProductionStatus() const;
```

Returns the current value of the production status maintained by the service instance.

CCSDS HISTORICAL DOCUMENT


```
SLE_YesNo Get_DirectiveInvocationOnline() const;
```

Returns yes when a service instance with directive invocation capability enabled is connected and 'no' otherwise.

```
FSP_BufferSize Get_PacketBufferAvailable() const;
```

Returns the current value of the parameter packet buffer available maintained by the service instance.

```
unsigned long Get_NumberOfADPacketsReceived() const;
```

Returns the accumulated number of AD packets received as derived from FSP-TRANSFER-DATA returns.

```
unsigned long Get_NumberOfBDPacketsReceived() const;
```

Returns the accumulated number of BD packets received as derived from FSP-TRANSFER-DATA returns.

```
unsigned long Get_NumberOfADPacketsProcessed() const;
```

Returns the accumulated number of AD packets processed including those for which processing was started but could not be completed and those for which start of processing was attempted.

```
unsigned long Get_NumberOfBDPacketsProcessed() const;
```

Returns the accumulated number of BD packets processed including those for which processing was started but could not be completed and those for which start of processing was attempted.

```
unsigned long Get_NumberOfADPacketsRadiated() const;
```

Returns the accumulated number of AD packets that were completely radiated.

```
unsigned long Get_NumberOfBDPacketsRadiated() const;
```

Returns the accumulated number of BD packets that were radiated.

```
unsigned long Get_NumberOfPacketsAcknowledged() const;
```

Returns the accumulated number of AD packets, for which processing was completed.

CCSDS HISTORICAL DOCUMENT

FSP_PacketId Get_PacketLastProcessed() const;

Returns the current value of the parameter packet identification last processed.

Precondition: either `Get_NumberOfADPacketsProcessed()` or `Get_NumberOfBDPacketsProcessed()` returns a non-zero number.

const ISLE_Time* Get_ProductionStartTime() const;

Returns the time at which processing of the packet identified by the parameter packet identification last processed started or was attempted. If no packets were processed yet, returns a NULL pointer.

FSP_PacketStatus Get_PacketStatus() const;

Returns the status of the packet identified by the parameter packet identification last processed. If no packets were processed yet, returns 'invalid'.

FSP_PacketId Get_PacketLastOk() const;

Returns the current value of the parameter packet identification last OK.

Precondition: either `Get_NumberOfADPacketsAcknowledged()` or `Get_NumberOfBDPacketsRadiated()` returns a non-zero number.

const ISLE_Time* Get_ProductionStopTime() const;

Returns the time at which the packet identified by packet identification last OK completed processing. If no packets completed processing yet, returns a NULL pointer.

FSP_PacketId Get_ExpectedPacketId() const;

Returns the packet identification expected next as derived from FSP-START and FSP-TRANSFER-DATA operations.

FSP_DirectiveId Get_ExpectedDirectiveInvocationId() const;

Returns the directive invocation identifier expected next as derived from FSP-INVOKE-DIRECTIVE operations.

FSP_EventInvocationId Get_ExpectedEventInvocationId() const;

Returns the event invocation identifier expected next as derived from FSP-THROW-EVENT operations.

Initial Parameter Values

Parameter	Value
production status	initial production status set via the interface `IFSP_SIAdmin`
directive invocation online	initial value set via the interface `IFSP_SIAdmin`
packet identification last processed	0
production start time	NULL
packet status	'invalid'
packet identification last OK	0
production stop time	NULL
packet buffer available	maximum packet buffer size set via the interface `IFSP_SIAdmin`
number of AD packets received	0
number of BD packets received	0
number of AD packets processed	0
number of BD packets processed	0
number of AD packets radiated	0
number of BD packets radiated	0
number of packets acknowledged	0
expected packet identification	0
expected directive invocation id	0
expected event invocation id	0

CCSDS HISTORICAL DOCUMENT

ANNEX B

ACRONYMS

(Informative)

This annex expands the acronyms used throughout this Recommended Practice.

API	Application Program Interface
CCSDS	Consultative Committee for Space Data Systems
CLCW	Communications Link Control Word
FOP	Frame Operation Procedure
FSP	Forward Space Packet
GUID	Globally Unique Identifier
ID	Identifier
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
MAP	Multiplexer Access Point
OMG	Object Management Group
PDU	Protocol Data Unit
RF	Radio Frequency
SI	Service Instance
SLDU	Space Link Data Unit
SLE	Space Link Extension
UML	Unified Modeling Language
VC	Virtual Channel
V(R)	Receiver Frame Sequence Number
V(S)	Transmitter Frame Sequence Number

CCSDS HISTORICAL DOCUMENT

ANNEX C

INFORMATIVE REFERENCES

(Informative)

[C1] *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.

[C2] *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.

[C3] *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.

[C4] *Space Link Extension—Internet Protocol for Transfer Services*. Recommendation for Space Data System Standards, CCSDS 913.1-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, September 2008.

[C5] *Space Link Extension—Application Program Interface for Transfer Services—Application Programmer's Guide*. Report Concerning Space Data System Standards, CCSDS 914.2-G-2. Green Book. Issue 2. Washington, D.C.: CCSDS, October 2008.

[C6] *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>

[C7] *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>