CCSDS Conformant Services (SLES)

SLE API Package

Reference Manual

ABSTRACT

This document provides the information required by software developers to integrate and use the SLE Package. It also provides information required for installing, configuring, and operating the communication infrastructure required by the API.

Document Change Record

Issue	Reason	Date
1.0	First Issue	20.04.1999
2.0	Issue at the end of the AD phase (includes updates from the SR/R)	16.08.1999
2.1	Changes reflecting the CCN, Updates from the AD/R, Changes in the C API type definitions	14.02.2000
3.0	Additions after the DD and API implementation phase	04.09.2000
3.1	Minor corrections of references	29.09.2000
3.2	Includes additions for the NT-4.0 support	15.12.2000
3.3	Includes additions for MS Windows 2000 support, API V1.4 updates	10.09.2001
3.4	Minor updates for API V1.5	20.02.2002
3.5	Minor corrections for API V1.6	28.03.2003
4.0	Support for API Version 2 added, Removed NT-4.0 support	29.07.2004
4.1	Minor corrections	10.03.2005
4.2	Updates following Release 3.0.0, Stub and Debug library support removed	16.02.2007
4.3	Added Information for Solaris 10 Support	05.03.2007
4.4	Removed C interface acronym for Stub and Debug	12.04.2007
4.5	Changes for new ASN1 product	06.06.2007
4.6	Reintroduced support of the Windows NT 4.0 platform	02.07.2008
4.7	Reintroduced support of the OpenVMS 7.1 AXP platform	11.08.2008
4.8	Includes Communication Server Library and SLE Version 3 with picoseconds support for the Earth Receive Time	30.10.2009
4.9	Issue for SLE API Version 3.5.0, SLE Service Version 4	28.05.2013
4.10	Issue for SLE API Version 3.6.2, SLE API upgrade to SLES12	28.04.2016

Table of Contents

1 Introduction

1.1 Intended Readership

This document is intended for software developers using the ESA SLE API package to build SLE applications. It is assumed that readers are familiar with the CCSDS Recommendations for Space Link Extension services (see references in section 1.6.1) and the specification of the SLE C++ Application Program Interface for Transfer Services (see references in section 1.6.2)

1.2 Applicability

This issue of the ESA SLE API Reference Manual is directly applicable to the ESA SLE API package

- Version 3.6 for Solaris 8
- Version 3.6 for SUSE Linux Enterprise 12.0 (64bit)
- Version 3.6 for SUSE Linux Enterprise 11.0 (32bit and 64bit)
- Version 3.6 for MS Windows 2003

and will continue to apply to all subsequent releases, until further notice. New issues of this Reference Manual will not necessarily coincide with software releases.

1.3 Purpose

This document provides the information required by software developers to integrate and use the ESA SLE package. It also provides information required for installing, configuring, and operating the communication infrastructure required by the API.

The ESA SLE API package is an implementation of the SLE C++ Application Program Interface for Transfer Services, specified in reference [SLE-API]. It provides an interface for SLE Applications to inter-operate, which is independent of any specific communications technology. For communication, the ESA SLE API package uses TCP/IP and the mapping of the SLE Protocol to TCP/IP specified in reference [TCP-PROXY].

It is assumed that both specifications are available to the reader. This document describes the ESA specific extensions to the SLE API specification and provides implementation specific information.

1.4 How to use this Document

This document is a reference manual and intends to provide detailed and complete information. With exception of chapter 2, providing an overview, the individual chapters and appendices should be consulted when needed during development of an application.

Readers not familiar with the SLE API specification should first consult chapter 2 of reference [SLE-API]. That chapter describes the concepts and the structure of the API, and introduces the terms used in this document. Before proceeding to the more detailed parts of this document (chapters 3 to 6), it is recommended to read chapter 3 of the SLE API specification, which presents an UML model of the API.

Chapter 2 provides a brief overview and defines the scope of the implementation by
- listing the options defined in the SLE API specification, that are supported;
- listing the SLE Transfer Service types that are supported.

Chapter 3 identifies the selected implementation approach used for the ESA SLE API where the SLE API specification allows implementers to choose.

Chapter 4 discusses specific requirements on the supported platforms.

Chapter 5 provides a detailed specification of the extensions to the SLE API, including
- a "Down Call Wrapper" that allows applications to retrieve events from the API instead of providing interfaces that are called by the API;
- a set of convenience functions for initialisation, control, and shutdown of the API;
- a special interface for logging and tracing of events that are not related to a service instance.

In addition, this chapter provides a discussion of the concepts for the C Language interface to the API. The specification of the C function prototypes can be found in Appendix C.

Chapter 6 provides all information needed to compile and link programs using the ESA SLE API package.

Chapter 7 describes how the ESA SLE API package is configured. It explains the format of the configuration files and provides a detailed specification of all configuration parameters.

Chapter 8 describes the data communication infrastructure required by the API package and provides (or references) the instructions for installation, configuration, and operation.

Appendix A provides supplementary information on specific issues related to the communications server process.

Appendix B lists the log messages that are issued by components of the ESA SLE API Package.

Appendix C describes the trace records that are generated by components of the ESA SLE API Package.

Appendix D lists all files in the distribution of the ESA SLE API Package.

It is recommended that application developers include the information provided by chapters 7 and 8 and by the appendices B and C into the documentation of SLE applications.

1.5 Definitions, Acronyms, and Abbreviations

1.5.1 Definitions

The terms used in this document are defined in reference [SLE-API].

1.5.2 Acronyms and Abbreviations

API	Application Program Interface
CCSDS	Consultative Committee for Space Data Systems
CLTU	Communication Link Transmission Unit
DCW	Down Call Wrapper
ESA	European Space Agency
ESOC	European Space Operations Centre
ESTRACK	ESA Tacking Network
FSP	Forward Space Packet
IP	Internet Protocol
ISO	International Standardisation Organisation
PDU	Protocol Data Unit
RAF	Return All Frames
RCF	Return Channel Frame
ROCF	Return Operational Control Fields
SE	Service Element
SHA	Secure Hash Algorithm
SI	Service Instance
SLE	Space Link Extension
SLES	SLE Services
TCP	Transmission Control Protocol

1.6 Related Documents

1.6.1 Space Link Extension Services

Space Link Extension Services are specified by the following CCSDS Recommendations.

Cf. SLE API Package References document [SLREF] for edition/revision of each document.

[CCSDS 910.4] *Cross Support Reference Model—Part 1, Space Link Extension Services*
Recommendation for Space Data System Standards,

[CCSDS 911.1] *Space Link Extension – Return All Frames Service Specification,*
Recommendation for Space Data System Standards,

[CCSDS 911.2] *Space Link Extension – Return Channel Frames Service Specification,*
Recommendation for Space Data System Standards,

[CCSDS-911.5] *Space Link Extension – Return Operational Control Field Service Specification,*
Recommendation for Space Data System Standards,

[CCSDS 912.1] *Space Link Extension – Forward CLTU Service Specification,*
Recommendation for Space Data System Standards,

[CCSDS 301.0] *Time Code Formats*. Recommendation for Space Data Systems Standards

1.6.2 SLE Application Program Interface

The SLE API and the mapping of the SLE protocol to TCP/IP are specified by the following documents.

[SLREF] *ESA SLE API Package - References, Acronyms and Abbreviations*, SL-ANS-REF-0002, Issue 3.4, 25.04.2016

[SLE-API] *Space Link Extension - Application Program Interface for Transfer Services – Core Specification,* Draft Recommended Practice, CCSDS 914.0-M-1, Issue 1, October 2008

[RAF-API] *Space Link Extension - Application Program Interface for Return All Frames Service*, CCSDS 915.1-M-1, Issue 1, October 2008

[RCF-API] *Space Link Extension - Application Program Interface for Return Channel Frames Service*, CCSDS 915.2-M-1, Issue 1, October 2008

[ROCF-API] *Space Link Extension - Application Program Interface for the Return Operational Control Fields Service,* CCSDS 915.5-M-1, Issue 1, October 2008

[CLTU-API] *Space Link Extension - Application Program Interface for the Forward CLTU Service*, CCSDS 916.1-M-1, Issue 1, October 2008

[FSP-API] *Space Link Extension - Application Program Interface for the Forward Space Packet Service*, CCSDS 916.3-M-1, Issue 1, October 2008

[TCP-PROXY] *Specification of a SLE API Proxy for TCP/IP and ASN.1*, SLES-SW-API-0002-TOS -GCI, Issue 1.2, May 2004

1.6.3 Unified Modelling Language

The Unified Modelling Language (UML) used in this document is specified by

[UML-1] *Unified Modelling Language (UML),* Version 1.5, Object Management Group, formal/2003-03-01, March 2003 (http://www.omg.org/technology/documents/modeling_spec_catalog.htm)

1.7 Conventions

This document uses the conventions defined in the SLE API specification.

Software design information is presented using the Unified Modelling Language (UML) and applying the conventions defined in section 3.2 of reference [SLE-API].

Interfaces of the extensions to the API are specified using the conventions defined in section 6.2 of reference [SLE-API].

2 Overview

2.1 Introduction

The ESA SLE API package is a portable implementation of the “SLE C++ Application Program Interface for Transfer Services”, specified in reference [SLE-API]. The SLE API provides an interface for SLE Applications to inter-operate, which is independent of any specific communications technology. For communication, the ESA SLE API Package uses TCP/IP and the mapping of the SLE protocol to TCP/IP specified in reference [TCP-PROXY]. Physically, the package consists of a set of C++ and C header files and set of libraries that are linked to the application program.

The ESA SLE API package implements the full set of API components specified in reference [SLE-API] i.e.

- the component API Service Element;
- the component API Proxy;
- the component SLE Operations;
- the component SLE Utilities.

A complete API can be built from components provided by this package. If needed, components of this package can also be combined with API components provided by other suppliers.[1]

The ESA SLE API package contains two versions of the API, one for SLE user applications and one for SLE provider applications.

It also extends the SLE API by providing

- an additional component, the "Down Call Wrapper", which allows applications to retrieve events from the API instead of providing interfaces that are called by the API;
- a set of convenience functions to construct, initialise, and control the API;
- a special interface for logging of events that are not related to a service instance;
- an additional API for the C Language.

An overview on the additional features can be found in section 2.2.

Section 2.3 describes the communications infrastructure used by the package.

Section 2.4 explains how the ESA SLE API can be configured.

Sections 2.5 and 2.6 define the scope of the implementation listing the optional features that are implemented and the SLE Transfer Services that are supported.

Section 2.7 concludes this overview with a description of the platforms for which the package is available and of the features that are supported on each platform.

[1] The SLE API Specification defines some optional features. In some cases, components not supporting all options cannot co-operate (see [SLE-API], chapter 8). The options supported by the ESA SLE API are listed in section 2.5

2.2 Extensions to the SLE API

2.2.1 The Down Call Wrapper Component

2.2.1.1 Purpose

The component Down Call Wrapper (DCW) provides an event-based interface by which the application can retrieve events by a call to the DCW instead of providing interfaces that are called by the API. The interface is suitable for use by single threaded applications and multi-threaded applications. Because the DCW uses only standard interfaces of the component API Service Element it can be used together with any API Service Element conforming to the SLE API Specification.

2.2.1.2 General Concepts

The concept of the Down Call Wrapper is illustrated in Figure 2-1. The DCW provides an interface (`IDCW_SIFactory`) for creation and deletion of service instances, which the application must use instead of the interface `ISLE_SIFactory` exported by the Service Element. The DCW uses the interface of the API Service Element to create the Service Instance object and links it with an internal DCW Service Instance object (DCW SI 1..N), which implements the interface `ISLE_ServiceInform`. This interface is used by the service instance in the component API Service Element to call the application.

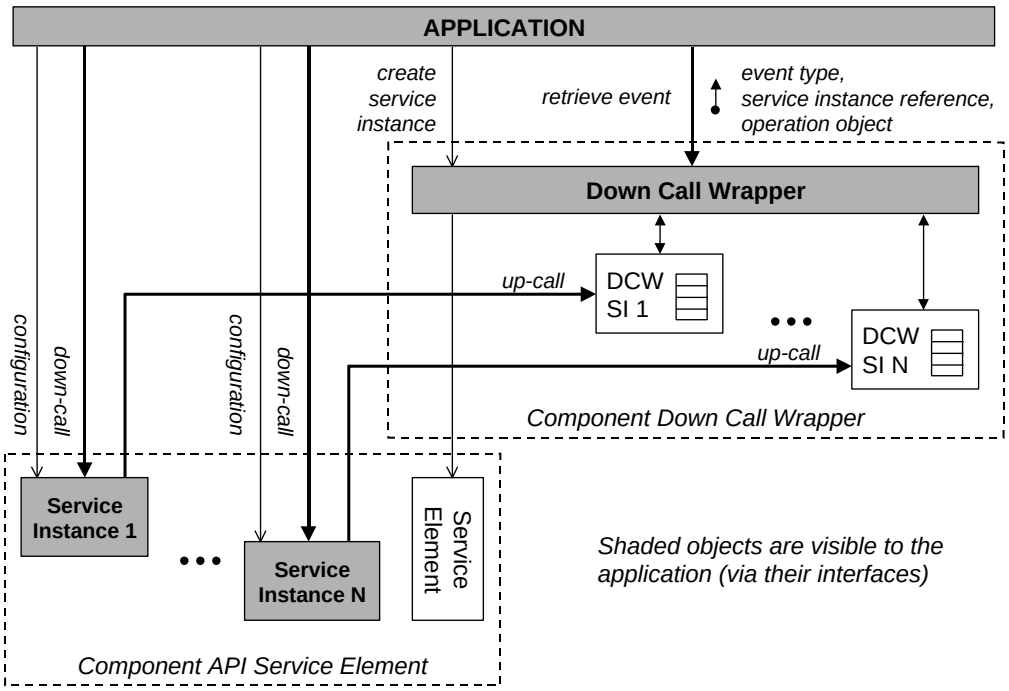

Figure 2-1 Down Call Wrapper

The method `IDCW_SIFactory:CreateServiceInstance()` for creation of a service instance object returns a reference to the standard service instance in the API Service Element. The application uses this reference to configure the service instance and to send operation invocations and returns via the standard interfaces defined in [SLE-API].

Operation objects and other events passed to the DCW Service Instance via the interface `ISLE_ServiceInform` are stored to a queue. The DCW exports an additional interface (`IDCW_EventQueue`) by which the application can retrieve these events by calling the DCW.

The methods for event retrieval return the following information:

- the type of event (e.g. operation invocation, operation return, protocol abort);
- a reference to the service instance in the API Service Element that generated the event;
- a reference to the operation object if the event refers to an operation invocation or return.

This information allows the application to associate the operation objects with the service instance to which they belong. The approach to allocate this task to the application has the advantage that the application can use the standard interfaces of the service instance as defined in reference [SLE-API]. Because the DCW does not handle these interfaces, addition of service type specific interfaces does not affect the DCW.

2.2.1.3 Event Handling

The DCW provides a platform specific "event handle"[2] (see [SLE-API], section 6.3), which is signalled whenever an event is pending. This can be used by the application to wait for events generated by the API in parallel to other external events it must process. The event handle can be retrieved via the interface `IDCW_EventQueue`.

For retrieval of events, the DCW provides a blocking method and a non-blocking method. The blocking method returns only when an event is present. The non-blocking method returns immediately when no event is currently queued.

Events generated by one service instance are delivered in the sequence they are received from the API Service Element. The only exceptions are the PEER-ABORT Invocation and the event protocol abort. When these events are received all events on the queue are flushed and the abort event is placed on the head of the queue.

The sequence, in which the DCW delivers events for different service instances, is, in principle, undefined. The DCW makes sure that events from different services are processed in a "fair manner". “Fair” is used in the sense that on average all active service instances are served with the same frequency.

The maximum number of events that are queued for one service instance can be defined when creating the service instance. If this number is exceeded, the DCW blocks delivery of events by the API such that backpressure across the network is built up.

2.2.1.4 Using the Service Instance Interfaces

The Down Call Wrapper provides an interface that can be used by a single threaded application and is safe in the presence of multiple threads. However, it uses the API Service Element with the behaviour "Concurrent Flows of Control" (see [SLE API], sections 3.3.4 and 4.7). Because the API Service Element provides and expects the same behaviour on all of its interfaces, this behaviour also applies to the interfaces used directly by the application.

In general, an application does not need to be aware of the fact that the interfaces of service Instances handle multiple threads, because the service instance never calls the application. However, the SLE API specification requires that sequence counting be supported for SLE Protocol Data Units when these are passed across an interface with "concurrent" behaviour. Consequently the application must correctly handle the sequence count argument in the methods of the interface `ISLE_ServiceInitiate`. If the application is single-threaded, sequence counting is trivial. All it requires is holding a variable for the sequence count for every service instance, setting it to one for the BIND invocation or return and subsequently incrementing it for every call to `InitiateOpInvoke` and `InitiateOpReturn`.

[2] Examples for event handles include a file descriptor on UNIX that can be used in the system call poll() or select().

2.2.1.5 Aborting Associations

Because the application uses the standard interfaces of the Service Instance object directly, the DCW will not be informed when the application aborts an association[3]. This can have the effect that events remain on the queue of the DCW Service Instance and cause the event handle to be signalled.

In order to prevent such behaviour, the interface `IDCW_EventQueue` provides a method to flush the queue for a specified service instance. This method should be called whenever the application does not expect any further events.

It is noted that the only task of the DCW Service Instance object is converting an "up-call" to an event that can be retrieved by the application. It is not aware of the state of the service instance. Therefore, it is the responsibility of the application to flush the queue only when this is appropriate.

2.2.1.6 Configuration and Control

The Down Call Wrapper is implemented as an API component, following the rules and conventions defined in Appendix A of reference [SLE-API]. It can be used together with other API components form other suppliers, if needed.

In order to allow use of components from various sources, the application must create, configure, and link all API components individually. This also applies for the Down Call Wrapper.

- it must be created using a special "creator function";
- it must be linked with the API Service Element using the method `Configure` in its administrative Interface.

Operation of the complete API must be controlled via the methods `Start` and `Terminate` in the administrative interface of the DCW.

If an application uses components of the ESA SLE API Package only, convenience functions are made available to perform these tasks (see section 2.2.2).

2.2.2 Convenience Features

The ESA SLE API Package provides a set of convenience functions to build, initialise, and control the API. These functions are provided by the classes `ELSE_APIBuilder` and `ELSE_Builder`, specified in section 5.2.

`ELSE_Builder`

As delivered, the class `ELSE_Builder` builds the API from components of the ESA SLE API Package and includes the Down Call Wrapper. If components from other sources shall be used the source code of this class must be modified, or the application must build the API itself.

With the class `ELSE_Builder`, the application only needs to handle the interfaces it really needs for communication with the API. For instance, it does not need to be aware of the existence of the API Proxy. The sequence of actions that an application must perform, are the following:

1) Creates and configures the complete API using the method `Initialise()`.

2) Obtains the interfaces of API components needed using the appropriate "Get" methods.

3) Starts operation of the API using the method `Start()`.

The application can now use the API as described in reference [SLE-API]. To close down, the application performs the following actions.

4) Terminates operation of the API using the method `Terminate()`.

5) Closes down and terminates the API using the method `ShutDown()`.

[3] Abort of an association might be required because of operator intervention or because of equipment failure.

In addition to the builder, the ESA SLE API Package provides multi-thread safe wrappers for the Reporter, Trace, and Time Source classes, which are also described in section 5.2.

`ELSE_APIBuilder`

In contrast to `ELSE_Builder`, the class `ELSE_APIBuilder` builds the API from components of the ESA SLE API Package and does **not** include the Down Call Wrapper. If components from other sources shall be used the source code of this class must be modified, or the application must build the API itself.

With the class `ELSE_APIBuilder`, the application only needs to handle the interfaces it really needs for communication with the API. The sequence of actions that an application must perform, are the following:

1) Creates and configures the complete API using the method `Initialise()`.

2) Obtains the interfaces of API components needed using the appropriate "Get" methods.

3) Starts operation of the API using the method `Start()`.

The application can now use the API as described in reference [SLE-API]. To close down, the application performs the following actions.

4) Terminates operation of the API using the method `Terminate()`.

Closes down and terminates the API using the method `ShutDown()`.

2.2.3 Special Logging Interface

The SLE API supporting the SLE responder role uses a separate communications server process, which listens for incoming connection requests and routs BIND invocations to different application processes (see section 2.3). The communications server process also routs log messages, notifications, and trace records that refer to specific service instances to the application process handling the service instance. In order to receive log messages and notifications about events that are not related to a specific service instance, the application can use a special interface referred to as the “default logger”. This interface can also be used to control tracing by the communications server process and to receive trace records.

The distribution contains a simple default logger program that can be used if the application does not support this interface. This program simply prints log messages to standard output.

2.2.4 C Application Program Interface

The ESA SLE API Package provides a SLE API for the C Language. The C-API is actually a small layer above the C++ API i.e. the C interface functions call the associated C++ interfaces.

The C-API preserves the object-oriented structure of the SLE API. References to object interfaces are returned by the C functions as opaque data types and must be passed to the functions of the C-API when a method of an interface shall be invoked. A more detailed description of the concepts can be found in section 5.4. The function prototypes are specified in Appendix C.

The C-API is provided only for those interfaces that must be used by the application. In addition, it requires use of the Down Call Wrapper and of the convenience features addressed in section 2.2.2. The interfaces supported by the C-API include:

- the convenience functions for building and controlling the API (see section 5.2);
- the interfaces the DCW provides to the application (see section 5.1);
- the interfaces a Service Instance provides to the application (see [SLE-API, section 6.8) ;
- the interfaces of common operation objects (see [SLE-API], section 6.5);
- the interfaces of SLE Utilities needed by the application[4] (see [SLE-API], section 6.4);
- the service type specific interfaces to configure service instances;
- the interfaces of service type specific operation objects;

[4] Utilities handling authentication are not needed by the application and do not provide a C interface.

- the special interface for logging and tracing described in section 5.3.

The SLE Services supported by the C-API include
- Return All Frames
- Return Channel Frames
- Forward Communication Link Transmission Unit

The Return Operational Control Fields and Forward Space Packet services are not supported by the C-API.

2.3 Communications Infrastructure

The ESA SLE API package uses TCP/IP for communications. The TCP and IP protocols are available as part of the operating system for all supported platforms. The operating systems also provide programs for configuration of these protocols.

An API supporting SLE user applications interfaces directly with TCP, and does not require any further infrastructure. An API supporting SLE provider applications uses a special TCP communications server process as indicated in Figure 2-2.

The communications server process must be started before any application starts the API. It is recommended to start it as a background (daemon) process after booting the operating system. When the API in an application process is started, it establishes an inter-process communication (IPC) channel to the communication server process. When the application creates and configures a service instance, the API registers the service instance identifier at the communications server process. The API de-registers the service instance identifier, when the application deletes the service instance.

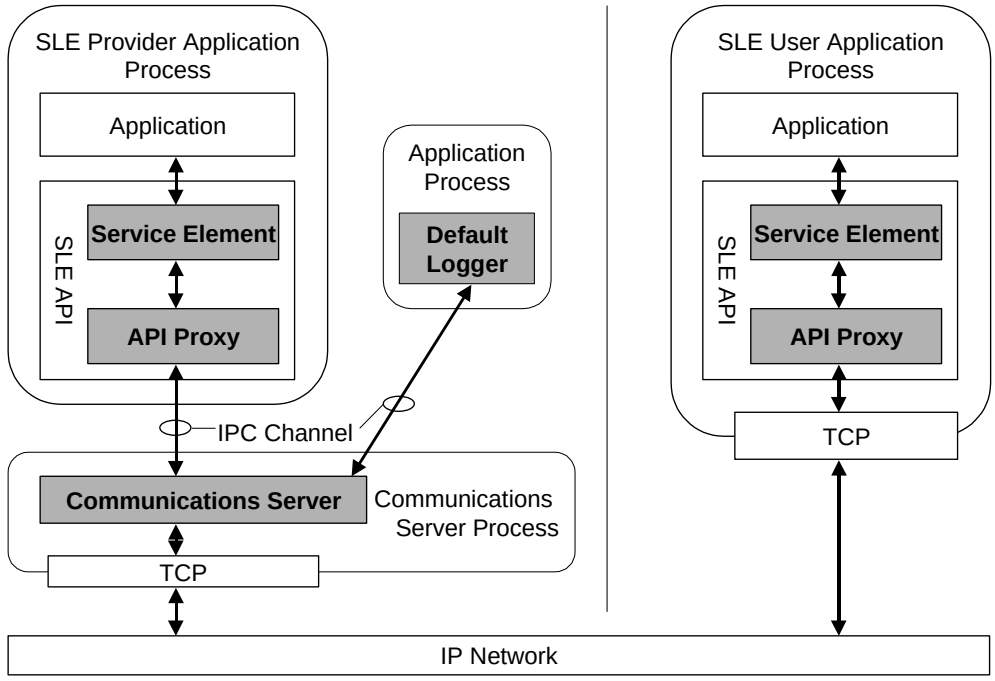

Figure 2-2 Communications Infrastructure for the ESA SLE API Package

The communication server process
- listens for incoming TCP connect requests;
- receives BIND invocations and routs them to the application process that has registered the service instance identifier identified in the BIND invocation PDU;
- receives SLE PDU's from the TCP connection and passes them to the proxy in the application process;
- accepts SLE PDU's from the proxy in the application process and transmits them via the TCP connection;

- sends log messages, notifications and trace records associated with a TCP connection to the proxy.

When receiving a BIND invocation with a service instance identifier that has not been registered, the communications server process rejects the BIND invocation with the appropriate diagnostics.

If an application wishes to receive log messages and notifications that are not related to a registered service instance, it must provide a process that receives these messages via the “default logger” interface. This can be a special process or a process using the API. Only one single instance in one single process can use this interface at a time.

APPENDIX A provides some further notes on specific issues related to the communications server process.

2.4 Configuration of the API

Configuration parameters for the ESA SLE API package are defined in text files using the format specified in chapter 7. The name of the configuration file must be passed to the API as part of its configuration.

Because the API Service Element and the API Proxy are individually substitutable, separate configuration files must be provided, which partially contain redundant information. Consistency of the information must be ensured when these files are prepared.

The communications server process on a SLE service provider system uses the same configuration file as the API proxy. The name of the file is passed to the process as a command line argument when the program is started.

2.5 Conformance to the SLE API Specification

The ESA SLE API package implements all mandatory features defined in reference [SLE API]. It is delivered in two versions, one supporting SLE user applications ("user version") and one supporting SLE provider applications ("provider version").

User Version

The user version supports the following options defined in section 8.3 of reference [SLE-API].

1) associations in the initiator role (option PXO-1a);
2) concurrent interface behaviour for the API Proxy (option PXO-2b);
3) diagnostic traces for the API Proxy (option PXO-5);
4) service instances for use by a SLE user application (option SEO-1a);
5) concurrent interface behaviour for the API Service Element (option SEO-3b);
6) diagnostic traces for the API Service Element (option SEO-4).

The option PXO-4 (routing of BIND invocations to different processes) does not apply, because it is only relevant for associations in the provider role. The options PXO-3 (operation mode for a gateway) and SEO-2 (provider initiated binding) are not supported.

Provider Version

The provider version supports the following options defined in section 8.3 of reference [SLE-API].

1) associations in the provider role (option PXO-1b);
2) concurrent interface behaviour for the API Proxy (option PXO-2b);
3) routing of BIND invocations to different processes (PXO-4);
4) diagnostic traces for the API Proxy (option PXO-5);
5) service instances for use by a SLE provider application (option SEO-1b);
6) concurrent interface behaviour for the API Service Element (option SEO-3b);
7) diagnostic traces for the API Service Element (option SEO-4).

The options PXO-3 (operation mode for a gateway) and SEO-2 (provider initiated binding) are not supported.

Limits

The SLE API specification defines a set of parameters that can be constrained by an implementation. The values supported by the ESA SLE API are defined in Table 2-1.

Parameter		**Supported Values**
1	Maximum number of concurrent bound associations per process	unconstrained
2	Maximum number of concurrent bound associations in total on a system	unconstrained
3	Maximum number of ports used for outgoing BIND Invocations per process.	unconstrained
4	Maximum number of ports on which an incoming BIND Invocation can be received per process	unconstrained
5	Maximum number of ports on which an incoming BIND Invocation can be received in total on a system	unconstrained
6	Maximum number of incoming PDU's that can be queued (per association / per process)	see note
7	Maximum number of incoming TRANSFER-DATA invocations and TRANSFER-BUFFER invocations that can be queued (per association / per process)	see note
8	Maximum number of outgoing PDU's that can be queued	unconstrained
9	Maximum number of pending remote Returns per association	unconstrained
10	Maximum size of a PDU	unconstrained
11	Maximum number of service instances that can exist concurrently	unconstrained
12	Maximum number of concurrently bound service instances	unconstrained
13	Maximum number of Proxies that can be supported concurrently	unconstrained
14	Maximum number of pending remote Returns per service instance	unconstrained
15	Maximum number of pending local Returns per service instance	unconstrained

Table 2-1 Supported Parameter Values

Note on parameters 6 and 7:

One incoming PDU can be queued per association. As defined in item 1, the number of associations per process is not constrained by the API software. The maximum number of queued PDU's per process is not constrained.

2.6 Supported SLE Transfer Services

The SLE Transfer Services specified in Table 2-2 are supported by the user version and by the provider version of the ESA SLE API package.

Note : version 3 was derived from the CCSDS Red Books

Service Name	Service Specification	API Specification
Return All Frames (RAF)	[CCSDS 911.1]	[RAF-API]

	Versions 1,2, 3* et 4	
Return Channel Frames (RCF)	[CCSDS 911.2] Versions 1, 2, 3* et 4	[RCF-API]
Return Operational Control Fields (ROCF	[CCSDS 911.5] Versions 1, 2 et 4	[ROCF-API]
Forward CLTU (CLTU)	[CCSDS 912.1] Versions 1, 2, 3* et 4	[CLTU-API]
Forward Space Packet (FSP)	[CCSDS 912.3] Versions 1, 2 et 4	[FSP-API]

Table 2-2 Supported SLE Transfer Services

* Version 3 appeared in CCSDS red books

2.7 Supported Platforms

The ESA SLE API Package is available on the platforms listed in Table 2-3 providing the APIs indicated.

Platform	User Version		Provider Version	
	C++ API	C-API	C++ API	C-API
Solaris 8	YES	YES	YES	YES
SUSE Linux 11.0 32 and 64 bit	YES	YES	YES	YES
SUSE Linux 12.0 64 bit	YES	YES	YES	YES
MS Windows 2003	YES	YES	YES	YES

Table 2-3 Supported Platforms

Note that the C-API only supports the RAF, RCF and CLTU services.

3 Implementation Specific Behaviour

3.1 Introduction

The SLE API specification [SLE-API] allows the implementer to choose specific behaviour. The following sections list and explain the behaviour chosen by the ESA SLE API implementation for specific issues.

3.2 Buffering and Flow Control for Return Link Services

The buffer used by the TRANSFER-BUFFER operation has a maximum size, which is set during service instance configuration.

In the delivery mode ‘online-timely’ the service instance requests the proxy to discard an already queued buffer when the new buffer is passed to the proxy. In this case the service instance inserts a SYNC-NOTIFY invocation indicating, “data discarded due to excessive backlog” at the beginning of the transfer buffer.

If the size of the buffer to send was already equal to the maximum size, this size becomes greater then the maximum size (insertion of the SYNC-NOTIFY invocation).

The SYNC-NOTIFY invocation indicating 'end of data' means that the data transfer is completed and the transfer buffer is sent. If the application passes data to the service element following a

notification 'end of data', the ESA SLE API does not check whether 'end-of-data' had been issued and accepts all further TRANSFER-DATA invocations.

3.3 Authentication failure

The following behaviour is needed to restrict the information that is made available to the peer system to the minimum possible. It conforms to the specifications in [TCP-PROXY].

- When authentication fails for a BIND return or an UNBIND invocation, the proxy passes a PEER-ABORT with the diagnostic "other reason" to the local client, and resets the underlying data communication.
- When authentication fails for a BIND invocation, the proxy resets the underlying data communication.
- When authentication fails for an UNBIND return, the proxy ignores it.
- When authentication fails for any other invocation or return, the proxy ignores it.

In all cases the API generates an authentication alarm.

3.4 UNBIND with the Reason 'end of provision period'

After reception of an UNBIND invocation with the reason 'end of provision period', the service instance is no longer accessible. This does not necessarily mean that it no longer exists. Subsequent BIND invocations to the same service instance can fail with different reasons.

If the service instance was already released by the application, the user will receive 'no such service instance' as diagnostic code in the BIND return. If the service instance still exists, the BIND invocation will be rejected with the diagnostic 'invalid time'.

3.5 Processing of CLTU Protocol Data Units

When the service element receives a CLTU Transfer-Data invocation or a CLTU Throw-Event invocation, it performs the checks defined in [SLE-API] and in [CLTU-API] SE-7 and SE-9. If any of the checks fail, the service element does not generate a return-PDU, but sets the result to "negative" and passes the operation invocation to the application.

This approach has been chosen for the ESA SLE API implementation, as it cannot guarantee that all return-arguments are set correctly, because the API does not know if all preceding PDUs have been already processed by the application.

This behaviour of the API conforms to the CLTU specific requirements SE-8 and SE-9 defined in reference [CLTU-API].

3.6 Processing of FSP Protocol Data Units

When the service element receives a FSP Transfer-Data invocation, a FSP Throw-Event invocation, or a FSP Invoke Directive invocation, it performs the checks defined in [SLE-API] and in [FSP-API] SE-21, SE-22, and SE-23. If any of the checks fail, the service element does not generate a return-PDU, but sets the result to "negative" and passes the operation invocation to the application.

This approach has been chosen for the ESA SLE API implementation, as it cannot guarantee that all return-arguments are set correctly, because the API does not know if all preceding PDUs have been already processed by the application.

This behaviour of the API conforms to the FSP specific requirements SE-20, SE-21, and SE-23 defined in reference [FSP-API].

3.7 Checking of BIND Invocations in the Provider role

The proxy notifies the service element of a BIND invocation received from the network through the interface `ISLE_Locator`.

When the service element receives a BIND invocation via the method `LocateInstance` of the interface `ISLE_Locator`, all checks on the BIND invocation are performed.

When the dedicated service instance receives the BIND invocation after service instance location, the checks on the BIND invocation are not repeated.

This implementation specific behaviour conforms to the specifications in SE-22 of reference [SLE-API].

3.8 Handling of Time Zones

CCSDS SLE Recommendations require that all time parameters be in UTC.

The ESA SLE API package assumes that the systems providing or using SLE Services will use UTC as their system time.

The ESA SLE API package makes use of the POSIX function `gmtime()` to convert the internal time representation to CCSDS time code formats. If an application uses a time zone other than UTC, times will be converted to UTC when generating time tags. The internal time used for timers remains unchanged.

3.9 Service Instance Provision Period

The ESA SLE API package implementation applies a timer for setting up the Service Instance Provision Period. There is an upper limit for the timeout imposed on timers by Solaris, which the API also applies on NT. On Solaris the maximum timeout that can be specified is 50,000,000 seconds which is 578.7 days or about 19 months.

In effect the API returns an error when the service provision period end is more than 19 month in the future.

3.10 Permanent Service Instances

According to reference [SLE-API], the method `Set_ProvisionPeriod()` of the service instance interface `ISLE_SIAdmin` allows setting of the provision period start and end times to zero, which indicates that the service instance provision period never ends. Reference [SLE-API] specifies that, if the start time is set to NULL, the service instance assumes immediate start of the provision period; if the stop time is NULL, the service instance provision period never expires.

For simplification reasons, the ESA SLE API implements the permanent service instance as follows:

If the start time is NULL, the service instance assumes immediate start of the provision period, and sets the start time to current time. If the stop time is NULL, the service instance provision period is ‘current time plus one year’.

3.11 Picoseconds Support

From SLE Version 3 onwards, the SLE API supports picoseconds resolution for the Earth Receive Time in the TRANSFER-DATA operation for return-link services.

The ESA SLE API supports picoseconds resolution as follows:

The ESA SLE API implementation of `ISLE_Time` allows setting the time with picosecond resolution for all SLE versions. It must be noted that the picosecond information will be cut when the time is to be encoded for transmission to the peer application for SLE Versions 1 and 2. In this case the time resolution is microseconds. The full picosecond resolution for the transmission of the Earth Receive Time is only supported for SLE Version 3.

In case the user wants the time in picosecond resolution although the `ISLE_Time` implementation has the time value only in microsecond resolution, then the time is returned in microsecond resolution, and the picoseconds are filled up with zeros.

4 Specific Requirements

4.1 Signal SIGPIPE on Unix

In order to minimise impact on the application, the ESA SLE API package does not generate signals, does not install any signal handlers, and does not modify the signal mask of calling threads.

Generation of the signal SIGPIPE by the operating system in case of premature closing of a socket or pipe cannot be prevented by the API.

In order to prevent termination of the process in such an event, it is recommended that the application use one of the following methods:

- Modify the signal mask of the initial thread to ignore SIGPIPE before calling the API. This is best done in the `main()` function by the following statement:

  ```
  sigignore(SIGPIPE);
  ```

- Install a signal handler for SIGPIPE before calling the API. The signal handler should ignore the signal if it was not caused by a file descriptor handled by the application.

5 Extensions to the SLE API

5.1 Down Call Wrapper Package

5.1.1 Overview

The Down Call Wrapper (DCW) package adds an event-based sequential interface to the SLE API to simplify integration of the SLE API into applications with event-loop architectures.

The DCW receives concurrent up-calls from several API Service Instances, via its `ISLE_ServiceInform` interfaces, and stores them in event queues. The client retrieves information sequentially from these queues via a single interface (`IDCW_EventQueue`) exported by the DCW. The DCW provides the client with a platform specific event-handle that the application may use to multiplex DCW events with other I/O.

The DCW always supplies events in the order that up-calls are received for any particular Service Instance, except for protocol-abort or PEER-ABORT Operation up-calls. When one of these occurs, the queue of the Service Instance is flushed and the abort event is placed on the head of the queue.

The client uses the `IDCW_SIFactory` to create one or more required Service Instances. The client interacts directly with the Service Instance through its `ISLE_SIAdmin`, `ISLE_SIOpFactory` and `ISLE_ServiceInitiate` interfaces (see [SLE-API] section 6.8), and uses the `IDCW_EventQueue` to retrieve events, which the Service Instance issues via calls to the interface `ISLE_ServiceInform`.

The DCW classes support logging and diagnostic traces using the interfaces `ISLE_Reporter` and `ISLE_Trace` provided by the application. Diagnostic traces can be switched on and off via the interface `ISLE_TraceControl` of the DCW, for tracing of the DCW, SLE API and all Service Instances. Tracing for individual Service Instances must be switched on and off via the interface of the Service Instance.

Note that the SLE application is responsible for creating and configuring of the DCW and SLE API components, but control of the Service Element and Proxy must be performed via the DCW.

The structure of the DCW is shown in Figure 5-1.

Application Obligations Satisfied

The DCW satisfies the following obligations of a SLE Application (reference [SLE-API] section 4.6.1):

SA-3	Implement `ISLE_ServiceInform`.
SA-18	Start processing of SLE API after configuration.
SA-20	Provide behaviour required by and interact with Service Element's `ISLE_Concurrent` interface.
SA-21	Create and delete service instances using `ISLE_SIFactory`.

Application Obligations Added

The DCW imposes additional obligations on its client application:

DCW-1	The application creates the DCW.
DCW-2	The application configures the DCW.
DCW-3	The application starts and terminates Service Element and Proxy processing via the DCW administrative interface only.
DCW-4	The application creates Service Instances via the `DCW_SIFactory` interface only.

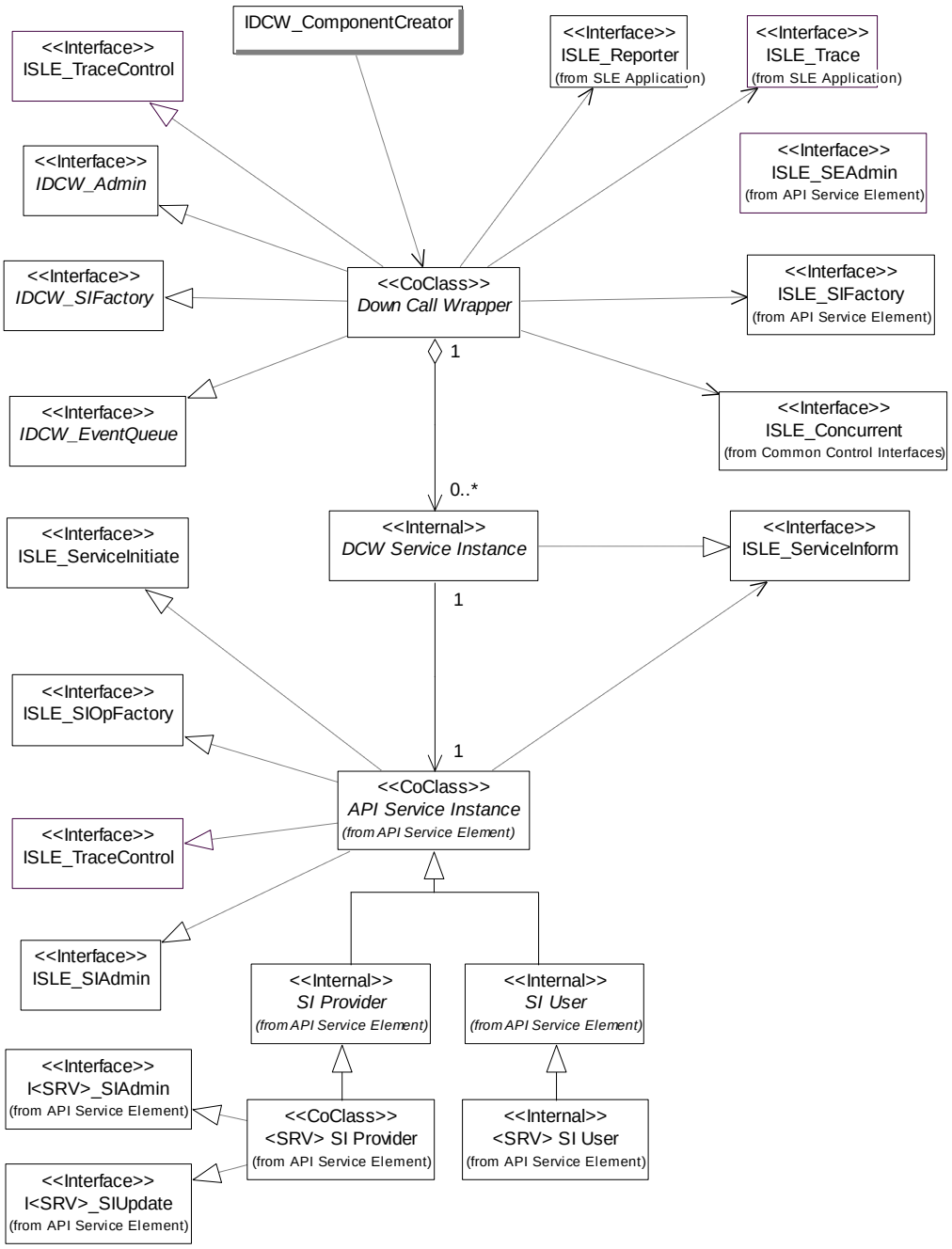

Figure 5-1 Down Call Wrapper Overview (Interfaces shown on the left-hand side of the figure are exposed for use by the application.)

5.1.2 Component Class Down Call Wrapper

The component class Down Call Wrapper provides overall management and control of the DCW and underlying Service Element and Proxy.

A single instance of this class exists within the Down Call Wrapper component.

Responsibilities

Configuration and initialisation of the DCW

Configured Service Element and Proxy(s) are registered with the DCW and the event queue initialised.

Control of the DCW, Service Element and Proxy

Management of the DCW event queue

The `IDCW_EventQueue` interface provides retrieval and removal of events from queues and signals queuing of events by the DCW Service Instances.

Management of DCW Service Instances

The class creates and deletes DCW Service Instances on request of the client via the interface `IDCW_SIFactory`. It uses the interface `ISLE_SIFactory` of the API Service Element to create Service Instance objects and links them with the DCW Service Instance.

Logging and Notification

Diagnostic Traces

Exported Interfaces

Interface	Defined in Package	Purpose
`IDCW_Admin`	Down Call Wrapper	Configuration, start, termination and shutdown of the DCW
`IDCW_SIFactory`	Down Call Wrapper	Creation & destruction of Service Instances
`IDCW_EventQueue`	Down Call Wrapper	Detection of and fetching DCW events
`ISLE_TraceControl`	Common Control Interfaces	Start and stop of diagnostic traces

Dependencies

Interface	Defined in Package	Purpose
`ISLE_Concurrent`	Common Control Interfaces	Start and control of the Service Element
`ISLE_SIFactory`	API Service Element	Creation & destruction of Service Instances
`ISLE_Reporter`	SLE Application	Logging and notification
`ISLE_Trace`	SLE Application	Tracing

5.1.3 Component Class DCW Service Instance

The component class DCW Service Instance is responsible for queuing up-calls from a Service Instance. To achieve this it implements the `ISLE_ServiceInform` interface that is used by the Service Instance.

An instance of this class exists within the Down Call Wrapper component for each instantiated API Service Instance.

Responsibilities

Queuing of up-calls from Service Instance

Linking of Service Instances with the DCW

Logging and Notification

Diagnostic Traces

Exported Interfaces

Interface	Defined in Package	Purpose
`ISLE_ServiceInform`	SLE Application	Queuing of SLE PDUs

Dependencies

Interface	Defined in Package	Purpose
`ISLE_Reporter`	SLE Application	Logging and notification
`ISLE_Trace`	SLE Application	Tracing

5.1.4 DCW Interfaces

5.1.4.1 DCW Types

File `DCW_Types.h`

In addition to the SLE API types, the DCW uses the following:

Event Type

```
typedef enum DCW_EventType
{
   dcwEVT_noEvent             = 0,
   dcwEVT_informOpInvoke      = 1,
   dcwEVT_informOpReturn      = 2,
   dcwEVT_resumeDataTransfer  = 3,
   dcwEVT_provisionPeriodEnds = 4,
   dcwEVT_protocolAbort       = 5
};
```

5.1.4.2 Component Creator Function

File DCW.H

The DCW package includes a function to create an instance and obtain a pointer to an interface of the DCW. The signature of this function is defined as:

```
extern "C" HRESULT ESLE_CreateDownCallWrapper(const GUID& dcwIid,
                                              void** pidcw);
```

Creates the Down Call Wrapper component and returns a pointer to the specified interface.

If the supplied GUID is not supported by the component, an error is returned. The function ensures that only one single instance of the DCW component is ever created. Every subsequent call to this function returns a pointer to an interface of the same instance.

Result Codes:

S_OK	the component has been instantiated
E_NOINTERFACE	the interface specified is not supported
E_FAIL	failure due to unspecified error

5.1.4.3 DCW Administrative Interface

Name IDCW_Admin
GUID {495008A0-F689-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File IDCW_Admin.H

This interface provides the means to configure and control the DCW component.

Clients use the method `Configure()` to register the other components with the DCW.

Processing in the DCW, Service Element and Proxy is initiated using the `Start()` method, and terminated using the `Terminate()` method. The Service Element and Proxy should be started and terminated via this interface only.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

interface ISLE_SEAdmin;
interface ISLE_UtilFactory;
interface ISLE_Reporter;

#define IID_IDCW_Admin_DEF { 0x495008a0, 0xf689, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface IDCW_DCWAdmin : IUnknown
{
  virtual HRESULT
    Configure( ISLE_SEAdmin* pse,
               ISLE_UtilFactory* putilFactory,
               ISLE_Reporter* preporter) = 0;
  virtual HRESULT
    Start() = 0;
  virtual HRESULT
    Terminate() = 0;
  virtual HRESULT
    Shutdown() = 0;
};
```

Methods

```
HRESULT Configure( ISLE_SEAdmin* pse,
                   ISLE_UtilFactory* putilFactory,
                   ISLE_Reporter* preporter );
```

Registers the Service Element, Reporter and Utility Factory with the DCW.

Arguments:

pse	pointer to the administrative interface of the Service Element
putilFactory	pointer to the Utility Factory interface to be used by the DCW
preporter	pointer to the Reporter interface for passing of log messages and notifications to the application from DCW

Result Codes:

S_OK	configuration completed without errors
SLE_E_STATE	already configured
E_FAIL	failure due to unspecified error

```
HRESULT Start();
```

Starts processing of the DCW, Service Element and Proxy. The DCW becomes ready for creation of Service Instances.

Result Codes:

S_OK	processing started
SLE_E_STATE	one or more of the components is not configured or is already started
E_FAIL	failure due to unspecified error

```
HRESULT Terminate();
```

Terminates processing of the DCW, Service Element and Proxy.

Bound Service Instances are aborted and deleted, and unprocessed DCW events are deleted. The event handle remains valid.

Result Codes:

S_OK	processing terminated
SLE_E_STATE	processing already terminated, or not yet started
E_FAIL	failure due to unspecified error

```
HRESULT Shutdown();
```

Releases all interfaces the DCW holds and deletes all internal objects. Note that the client must also release any interfaces it references (before or after this call) for the objects to be deleted.

The Event Handle will be invalid after `Shutdown`.

Processing must be explicitly terminated before `Shutdown` is called.

Result Codes:

S_OK	DCW & SLE-API interfaces released
SLE_E_STATE	not terminated
E_FAIL	failure due to unspecified error

5.1.4.4 DCW Service Instance Factory Interface

Name IDCW_SIFactory
GUID {E212DC53-E686-4977-A9A8-3B46E0EE4450}
Inheritance: IUnknown
File IDCW_SIFactory.H

This interface is used to create and destroy Service Instances when using the DCW.

The CreateServiceInstance method creates a Service Instance of the specified service-type and role and returns a pointer to its specified interface. The Service Instance is associated with a DCW Service Instance, which converts the Service Instance up-calls into DCW events.

It is the responsibility of the client to configure the Service Instance using the ISLE_SIAdmin interface and its service-type specific configuration interface (see [SLE-API] Section 6.8.5).

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

#define IID_IDCW_SIFactory_DEF {0xe212dc53, 0xe686, 0x4977, \
          { 0xa9, 0xa8, 0x3b, 0x46, 0xe0, 0xee, 0x44, 0x50 } }

interface IDCW_SIFactory : IUnknown
{
  virtual HRESULT
    CreateServiceInstance( const GUID& iid,
                           SLE_ApplicationIdentifier serviceType,
                           SLE_VersionNumber version,
                           SLE_AppRole role,
                           unsigned int maxPendingEvents,
                           void** psi) = 0;
   virtual HRESULT
      DestroyServiceInstance(IUnknown* psi) = 0;
};
```

Methods

```
HRESULT CreateServiceInstance( const GUID& iid,
                          SLE_ApplicationIdentifier serviceType,
                                SLE_VersionNumber version,
                                SLE_AppRole role,
                                unsigned int maxPendingEvents,
                                void** psi);
```

Creates a Service Instance of the specified service-type and role and returns a pointer to its specified interface.

Arguments:

iid	the identifier of the required Service Instance interface
serviceType	the SLE service-type to be supported by the Service Instance
version	for the user (initiator role) defines the version number of the SLE service type to be used; for the provider (responder) side this argument is ignored and should be set to zero.
role	the role (user or provider) of the required Service Instance
maxPendingEvents	maximum number of pending events for the Service Instance before up-calls from it are blocked
psi	pointer to the required interface of the created Service Instance

Result Codes:

S_OK	Service Instance created
SLE_E_UNKNOWN	the Service Instance is unknown
SLE_E_BADVALUE	maxPendingEvents is <1
SLE_E_STATE	processing in the DCW has not been started
SLE_E_INVALIDID	the version number is zero for the role “user”
E_FAIL	failure due to unspecified error

HRESULT DestroyServiceInstance(IUnknown* psi);

Destroys the Service Instance that owns the given interface together with the associated DCW Service Instance. All pending events from the Service Instance are removed from the event queue.

The method returns an error if the Service Instance is not in the unbound state.

Arguments:

psi	pointer to an interface of the created Service Instance

Result Codes:

S_OK	Service Instance destroyed
SLE_E_STATE	Service Instance is not in the unbound state
E_FAIL	failure due to unspecified error

5.1.4.5 DCW Event Queue Interface

Name IDCW_EventQueue
GUID {495008A3-F689-11d2-9B44-00A0246D80DB}
Inheritance: IUnknown
File IDCW_EventQueue.H

This interface is used to detect and fetch events from the DCW.

The DCW Service Instance queues an event whenever it receives an up-call, via its ISLE_ServiceInform interface, from a Service Instance. The client reads and removes the event from the head of the queue using either the NextEvent (blocking) or PollEvent (non-blocking) method. The client uses the EventType to determine which up-call was received on which Service Instance and to retrieve the Operation object, if applicable.

The Get_EventHandle method returns an event handle that is set whenever an event is present in any of the Service Instance queues. This enables the application to multiplex DCW event detection with other I/O.

If events are occurring faster than the application can process them they will accumulate in the queue, of a Service Instance. If the configured maximum queue-size is exceeded then the Service Instance adding the event will block until there is space. This in turn will cause the association to block which, if blocked for long enough, would time-out leading to peer abort by the peer application. Note that, depending on the implementation of the SLE-API, it is possible that all Service Instances will block.

The application can suspend the notification and reception of events by calling Suspend. No event except the reception of a PEER-ABORT will be signalled anymore. Any attempt to read an event with NextEvent or PollEvent will not return an event in this state. To enable the reception of events after a previous Suspend call the application calls Resume. An application can use the suspend / resume mechanism to build up backpressure to the sending site.

The FlushQueue method must be used immediately after the client has invoked a PEER-ABORT Operation on a Service Instance to remove any events pending for that Service Instance. After the FlushQueue, all the operations and events up to the next Bind operation will be discarded.

The EventType identifies the ISLE_ServiceInform up-call received and the base type of the operation passed, if any, as shown in Table 5-1. dcwEVT_noEvent represents absence of any event.

Up-call Method	Event Type	Operation Type
None	dcwEVT_noEvent	None
InformOpInvoke	dcwEVT_informOpInvoke	ISLE_Operation
InformOpReturn	dcwEVT_informOpReturn	ISLE_ConfirmedOperation
ResumeDataTransfer	dcwEVT_resumeDataTransfer	None
ProvisionPeriodEnds	dcwEVT_provisionPeriodEnds	None
ProtocolAbort	dcwEVT_protocolAbort + log Record	None

Table 5-1: Mapping of up-calls to Event Type

The ProtocolAbort up-call of the ISLE_ServiceInform interface delivers a diagnostic as a byte-array. This diagnostic is not delivered to the application as part of the event but is logged using the LogRecord method of ISLE_Report.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>
#include <DCW_Types.h>

interface ISLE_Operation;

#define IID_IDCW_EventQueue_DEF { 0x495008a3, 0xf689, 0x11d2, \
          { 0x9b, 0x44, 0x0, 0xa0, 0x24, 0x6d, 0x80, 0xdb } }

interface IDCW_EventQueue : IUnknown
{
  virtual SLE_EventHandle
    Get_EventHandle() = 0;
  virtual HRESULT
    NextEvent( DCW_EventType& eventType,
               ISLE_IUnknown** psi,
               ISLE_Operation** pop) = 0;
  virtual HRESULT
    PollEvent( DCW_EventType& eventType,
               ISLE_IUnknown** psi,
               ISLE_Operation** pop) = 0;
  virtual HRESULT
    FlushQueue( ISLE_IUnknown* psi ) = 0;
  virtual HRESULT Suspend () = 0;
  virtual HRESULT Resume () = 0;
};
```

Methods

`SLE_EventHandle Get_EventHandle();`

Returns an event handle that permits the application to multiplex DCW event detection with other I/O.

Preconditions:

The DCW, Service Element and Proxy have already been configured.

```
void NextEvent( DCW_EventType& eventType,
                IUnknown** psi,
                ISLE_API_Operation** pop);
```

Removes the next pending event and returns its type, the related Service Instance and Operation pointer. If the queue is suspended SLE_E_SUSPENDED will be returned.

The call blocks (i.e. does not return) until an event is available.

If the call fails the `eventType` will be `dcwEVT_noEvent`.

Arguments:

`eventType`	type of event
`psi`	pointer to the Service Instance related to the event
`pop`	pointer to the interface of the Operation related to the event, if applicable with `eventType`, else NULL

Result Codes:

`S_OK`	event removed from queue and returned
`SLE_E_STATE`	processing is not started
`E_FAIL`	failure due to unspecified error
`SLE_E_SUSPENDED`	queue is suspended

```
HRESULT PollEvent( DCW_EventType& eventType,
                   IUnknown** psi,
                   ISLE_API_Operation** pop);
```

Checks for a pending event and, if present, removes it from the queue, returning its type, Service Instance pointer and Operation pointer. If the queue is suspended S_FALSE will be returned.

If no event is present or if the call fails the `eventType` will be `dcwEVT_noEvent`.

Preconditions:

Arguments:

`eventType`	type of event
`psi`	pointer to the Service Instance related to the event
`pop`	pointer to the interface of the Operation related to the event, if applicable with `eventType`, else NULL

Result Codes:

`S_OK`	event removed from queue and returned
`S_FALSE`	no event available
`SLE_E_STATE`	processing is not started
`E_FAIL`	failure due to unspecified error

```
HRESULT FlushQueue( IUnknown* psi );
```

Removes any events from the queue of the specified Service Instance. After the `FlushQueue`, all the operations and events up to the next Bind operation will be discarded.

Arguments:

`psi`	pointer to the Service Instance

Result Codes:

`S_OK`	events removed from queue
`S_FALSE`	no events available for removal
`SLE_E_STATE`	processing is not started
`E_FAIL`	failure due to unspecified error

HRESULT Suspend();

Suspend the reception of operations except PEER ABORT operations for this queue. This applies to all service instances.

Result Codes:

S_OK	suspension sussessful
E_FAIL	suspension failed

HRESULT Resume();

Resume the reception of operations for this queue. This applies to all service instances.

Result Codes:

S_OK	resume sussessful
E_FAIL	resume failed

5.2 API Builder package

5.2.1 Overview

The API Builder package adds convenience classes to the DCW and SLE API to simplify integration of the DCW and SLE API with new and existing applications.

It provides:

- methods to initialise and shutdown a SLE application that uses only ESA SLE components and DCW
- MT-safe wrappers of the SLE API reporting and tracing interface methods
- Communication Server Library with on-line configuration update

The initialise and shutdown methods are implemented for applications using the DCW and ESA SLE API components only, but the source code is easily adaptable for using other implementations.

The SLE API requires that the methods exported by `ISLE_Reporter` and `ISLE_Trace` are MT-safe. The API Builder package provides help in implementing these by providing base-classes that the application specialises to implement the actual reporter and trace methods. The `Create_ISLE_<>` methods of the base-classes return the interface that uses the specialisation in a MT-safe way for its implementation.

There are applications, which might want to use their own Time Source. This is in general possible by passing the Time Source interface `ISLE_TimeSource` to the SLE API. The communication server, which is a separate process, does not have access to the Time Source interface in the application process. Therefore the communication server is also available as a library via the class `ELSE_CommunicationServer`, which enables an application to use its own time source, tracing and default logging interfaces on the communication server. A separate communication server process does not need to be started in this case.

Application Obligations Satisfied

The DCW and SLE API impose certain obligations in their client application, ref. section 5.1.1 and [SLE-API] section 4.6.1 respectively. The API Builder package provides functionality to satisfy the following obligations, in addition to those satisfied by the DCW (see section 5.1.1):

DCW-1 Create the DCW.

DCW-2 Configure the DCW.

DCW-3 Start and terminate Service Element and Proxy processing via the DCW administrative interface only.

SA-1 Create and configure the SLE API components.

SA-7 Perform orderly shutdown of SLE API.

SA-9 Provide MT-safe `ISLE_Reporter`.

SA-15 Provide MT-safe `ISLE_Trace`.

SA-16 Provide MT-safe `ISLE_TimeSource` interface

SA-17 Create the SLE API components needed for standard ESA installation.

SA-18 Configure and link together the Service Element and Proxy.

SA-19 Start processing of SLE API after configuration.

SA-20 Perform shutdown in specified sequence.

Application Obligations Added

The API Builder package imposes no additional obligations on its client application.

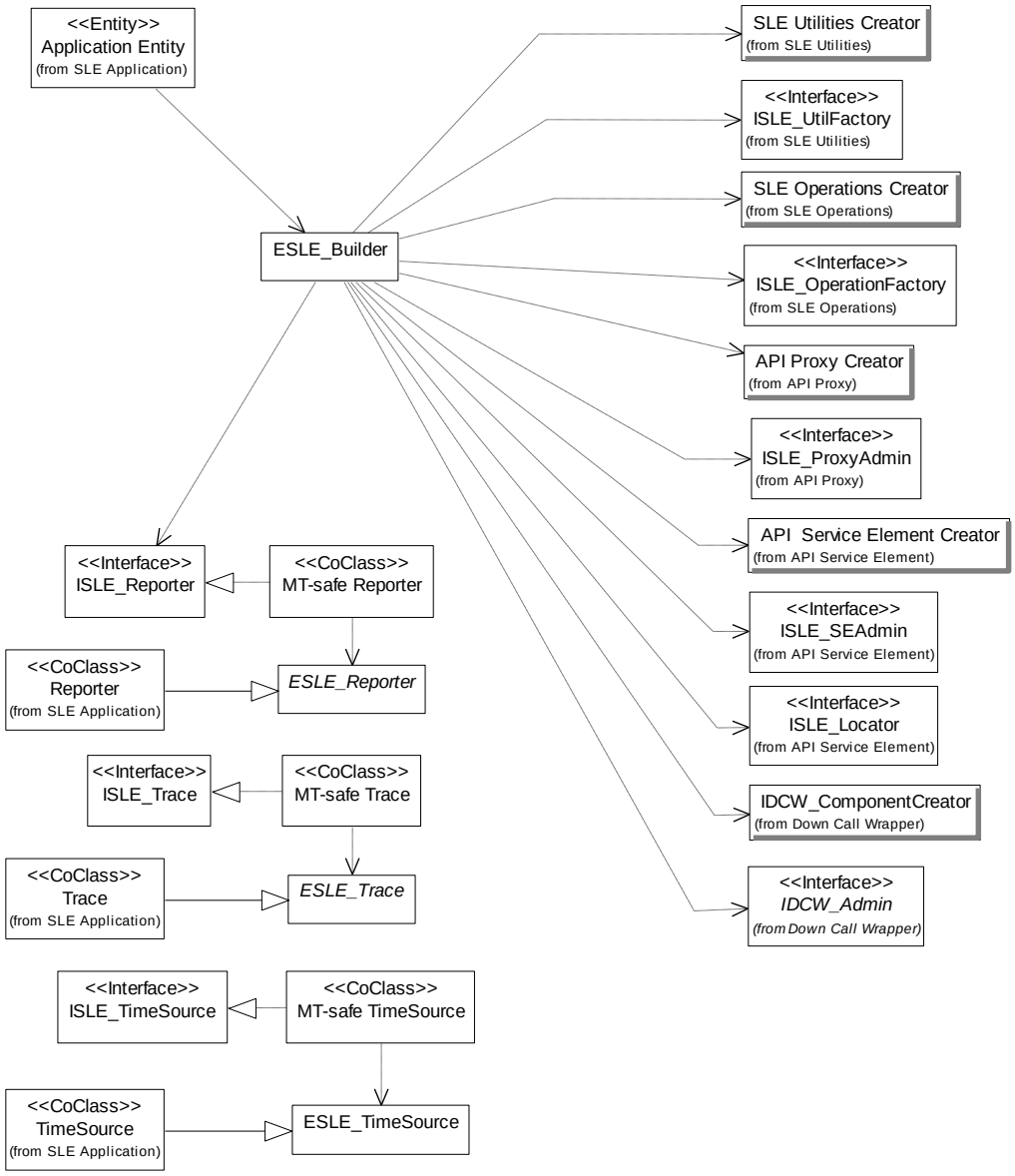

Figure 5-2 API Builder Overview

5.2.2 Class Builder

Name `ESLE_Builder`

Inheritance: `None`

File `ELSE_Builder.H`

This is a standard C++ class and not an interface following the conventions defined in [SLE-API], Appendix A.

This class provides methods to instantiate, configure and shutdown a SLE application that uses the DCW and ESA SLE API components.

The interface is a singleton accessed via the class method `Get_ESLE_Builder`.

First the DCW and SLE components are instantiated and configured by calling `Initialise`.

Then API processing is started by* calling `Start`. The components are now ready for Service Instance creation.

The required Service Instances are created and configured using the `IDCW_SIFactory`, `ISLE_SIAdmin` and service specific configuration interfaces.

Then, for a SLE user application, the application creates and invokes a BIND invocation using the Service Instance's `ISLE_SIOpFactory` and `ISLE_ServiceInitiate` interface and waits for the return using the `IDCW_EventQueue` interface. It then continues a dialogue with the service provider, using the Service Instance's `ISLE_ServiceInitiate` interface to send invocations and returns to the service provider, and `IDCW_EventQueue` to receive PDUs from the provider.

For a SLE provider application, the application waits for a BIND invocation using the `IDCW_EventQueue` interface, processes the bind and replies using the Service Instance's ISLE`_ServiceInitiate` interface. It then continues a dialogue with the service user, using `IDCW_EventQueue` to receive invocations and returns from the service user, and the Service Instance's `ISLE_SIOpFactory` and `ISLE_ServiceInitiate` interfaces to transfer PDUs to the service user.

When processing is finished the application stops processing within the ESA SLE API by calling the `Terminate` method. This will abort and destroy any bound Service Instances too.

Finally the application deletes the ESA SLE API components using the `Shutdown` method.

After a `Shutdown`, it is assumed that the application process will terminate. Thus it is not possible to reinitialise and restart the ESA SLE API after a `Shutdown`.

On the contrary, it is possible to restart the ESA SLE API by calling `Start` after a call to `Terminate`. Then the ESA SLE API will restart processing.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

interface ISLE_UtilFactory;
interface ISLE_Reporter;
interface IDCW_SIFactory;
interface IDCW_EventQueue;

class ESLE_Builder
{
public:
  static ESLE_Builder&
    Get_ESLE_Builder();
  HRESULT
    Initialise( const char* SEconfigFilePath,
                const char* proxyConfigFilePath,
                ISLE_Reporter* preporter,
                SLE_BindRole bindRole,
                ISLE_TimeSource* ptimeSource = 0 );
  HRESULT
    Start();
  HRESULT
    Terminate();
  HRESULT
    Shutdown();
  IDCW_SIFactory*
    Get_SIFactory() const;
```

```
  IDCW_EventQueue*
    Get_EventQueue() const;
  ISLE_UtilFactory*
    Get_UtilFactory() const;
};
```

Methods

`ESLE_Builder& Get_ESLE_Builder();`

Class method that returns the `ESLE_Builder` singleton object.

```
HRESULT Initialise( const char* SEconfigFilePath,
                    const char* proxyConfigFilePath,
                    ISLE_Reporter* preporter,
                    SLE_BindRole bindRole,
                    ISLE_TimeSource* ptimeSource = 0 );
```

Creates and configures the Down Call Wrapper, ESA Service Element, ESA TCP/IP Proxy, ESA SLE Utilities and ESA SLE Operations components.

Arguments:

`SEconfigFilePath`	path of the Service Element configuration file
`ProxyConfigFilePath`	path of the Proxy configuration file
`preporter`	pointer to a MT-safe Reporter interface for passing of log messages and notifications from the ESA SLE API
`bindRole`	indicates which role shall be used for the API.
`ptimeSource`	pointer to time source interface to be used to obtain the current time. If the argument is not supplied or zero, the current time is obtained from the system time.

Result Codes:

`S_OK`	initialisation & configuration completed without errors
`SLE_E_NOFILE`	configuration data file not found
`SLE_E_CONFIG`	error or inconsistency in configuration data
`SLE_E_STATE`	already initialised
`E_FAIL`	failure due to unspecified error

`HRESULT Start();`

Starts processing in the ESA SLE API. The DCW becomes ready for creation of Service Instances.

Result Codes:

`S_OK`	processing started
`SLE_E_STATE`	not initialised or already started
`E_FAIL`	failure due to unspecified error

`HRESULT Terminate();`

Stops processing in the ESA SLE API.

Service Instances are stopped and released and unprocessed DCW events are deleted, but the DCW event handle is still valid.

Result Codes:

`S_OK`	processing terminated
`SLE_E_STATE`	not initialised or already terminated
`E_FAIL`	failure due to unspecified error

HRESULT Shutdown();

Invokes the method `Shutdown` on all components. Note that the client must also release any interfaces it references (before or after this call) for the objects to be deleted. The DCW Event Handle will be invalid after `Shutdown`. Processing must be explicitly terminated before `Shutdown` is called.

Result Codes:

`S_OK`	the API has been shut down
`SLE_E_STATE`	not terminated
`E_FAIL`	failure due to unspecified error

IDCW_SIFactory* Get_SIFactory() const;

Returns the DCW interface for creating and destroying service instances (or NULL if not initialised). Service instances can not be created if processing has not been started.

IDCW_EventQueue* Get_EventQueue() const;

Returns the DCW interface for retrieving events from the DCW (or NULL if not initialised).

ISLE_UtilFactory* Get_UtilFactory() const;

Returns the Utility Factory interface (or NULL if not initialised), used for creating utility objects such as Time and Service Instance Identifier.

5.2.3 Class API Builder

Name `ESLE_APIBuilder`

Inheritance: `None`

File `ELSE_APIBuilder.H`

This is a standard C++ class and not an interface following the conventions defined in [SLE-API], Appendix A.

This class provides methods to instantiate, configure and shutdown a SLE application that uses ESA SLE API components, but does **not** use the Down Call Wrapper. When the DCW is not used, the application receives SLE operations and events via the interface `ISLE_ServiceInform`,

The interface is a singleton accessed via the class method `Get_ESLE_APIBuilder`.

The application first calls `Initialise`, which instanntiates and configures the SLE components.

Then the application starts API processing is by calling `Start`. The components are now ready for Service Instance creation using the `ISLE_SEAdmin`.

The required Service Instances are created and configured by the application, using the `ISLE_SIAdmin` and service specific configuration interfaces.

After successful creation and configuration of a service instance, a SLE user application creates and invokes a BIND invocation using the Service Instance's `ISLE_SIOpFactory` and `ISLE_ServiceInitiate` interface and waits for the return operation; this is passed to the application via the interface `ISLE_ServiceInform` (to be implemented by the application). The application can then send operations and returns to the service provider, using the Service Instance's `ISLE_ServiceInitiate` interface.

A SLE provider application waits for a BIND invocation, which is passed to the application via the interface `ISLE_ServiceInform` (to be implemented by the application); the application processes the bind operation and replies using the Service Instance's `ISLE_ServiceInitiate` interface. The provider application can then receive operation

invocations and can create and send operations to the user, using the Service Instance's `ISLE_SIOpFactory` and `ISLE_ServiceInitiate` interfaces.

When processing is finished, the application stops processing within the ESA SLE API by calling the `Terminate` method on the `ELSE_APIBuilder`. This will abort and destroy any bound Service Instances.

Finally the application deletes the ESA SLE API components using the `Shutdown` method.

After a `Shutdown`, it is assumed that the application process will terminate. Thus it is not possible to reinitialise and restart the ESA SLE API after a `Shutdown`.

On the contrary, it is possible to restart the ESA SLE API by calling `Start` after a call to `Terminate`. Then the ESA SLE API will restart processing.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

interface ISLE_UtilFactory;
interface ISLE_OperationFactory;
interface ISLE_SIFactory;
interface ISLE_SEAdmin;
interface ISLE_ProxyAdmin;
interface ISLE_Reporter;
interface ISLE_TimeSource;


class ESLE_APIBuilder
{
public:
  static ESLE_APIBuilder&
    Get_ESLE_APIBuilder();
  ISLE_OperationFactory*
    Get_OperationFactory() const;
  ISLE_ProxyAdmin*
    Get_ProxyAdmin();
  ISLE_SEAdmin*
    Get_SEAdmin();
  ISLE_SIFactory*
    Get_SIFactory() const;
  ISLE_UtilFactory*
    Get_UtilFactory () const;
  HRESULT
    Initialise( const char* SEconfigFilePath,
                const char* proxyConfigFilePath,
                ISLE_Reporter* preporter,
                SLE_BindRole bindRole,
                ISLE_TimeSource* ptimeSource = 0 );
  HRESULT
    Start();
  HRESULT
    Terminate();
  HRESULT
    Shutdown();
};
```

Methods

`ESLE_APIBuilder& Get_ESLE_APIBuilder();`

Class method that returns the `ESLE_Builder` singleton object.

```
HRESULT Initialise( const char* SEconfigFilePath,
                    const char* proxyConfigFilePath,
                    ISLE_Reporter* preporter,
                    SLE_BindRole bindRole,
                    ISLE_TimeSource* ptimeSource = 0 );
```

Creates and configures the ESA SLE API Service Element, ESA SLE API TCP/IP Proxy, ESA SLE API Utilities and ESA SLE API Operations components.

Arguments:

`SEconfigFilePath`	path of the Service Element configuration file
`ProxyConfigFilePath`	path of the Proxy configuration file
`preporter`	pointer to a MT-safe Reporter interface for passing of log messages and notifications from the ESA SLE API
`bindRole`	indicates which role shall be used for the API.
`ptimeSource`	pointer to time source interface to be used to obtain the current time. If the argument is not supplied or zero, the current time is obtained from the system time.

Result Codes:

`S_OK`	initialisation & configuration completed without errors
`SLE_E_NOFILE`	configuration data file not found
`SLE_E_CONFIG`	error or inconsistency in configuration data
`SLE_E_STATE`	already initialised
`E_FAIL`	failure due to unspecified error

`HRESULT Start();`

Starts processing in the ESA SLE API. After successful completion, the SLE API is ready for operation, creation of Service Instances can begin.

Result Codes:

`S_OK`	processing started
`SLE_E_STATE`	not initialised or already started
`E_FAIL`	failure due to unspecified error

`HRESULT Terminate();`

Stops processing in the ESA SLE API.

Service Instances are stopped and released. No further SLE operations can be sent.

Result Codes:

`S_OK`	processing terminated
`SLE_E_STATE`	not initialised or already terminated
`E_FAIL`	failure due to unspecified error

`HRESULT Shutdown();`

Invokes the method `Shutdown` on all components. Note that the client must also release any interfaces it references (before or after this call) for the objects to be deleted. Processing must be explicitly terminated before `Shutdown` is called.

Result Codes:

`S_OK`	the API has been shut down
`SLE_E_STATE`	not terminated
`E_FAIL`	failure due to unspecified error

`ISLE_SIFactory* Get_SIFactory() const;`

Returns the service instance interface for creating and destroying service instances (or NULL if not initialised). Service instances can not be created if processing has not been started.

`ISLE_UtilFactory* Get_UtilFactory() const;`

Returns the Utility Factory interface (or NULL if not initialised), used for creating utility objects such as Time and Service Instance Identifier.

`ISLE_OperationFactory* Get_OperationFactory() const;`

Returns the Operation Factory interface (or NULL if not initialised), used for creating operation objects.

`ISLE_SEAdmin* Get_SEAdmin() const;`

Returns the Service Element Administrative Interface (or NULL if not initialised). This interface is not needed if `ELSE_APIBuilder` is used for API initialisation, unless an additional proxy other than the ESA SLE API Proxy shall be added.

`ISLE_Proxydmin* Get_ProxyAdmin() const;`

Returns the Proxy Administrative Interface (or NULL if not initialised). This interface is not needed if `ELSE_APIBuilder` is used for API initialisation, unless an additional proxy other than the ESA SLE API Proxy shall be added via the `ISLE_SEAdmin` interface.

5.2.4 Class Reporter

Name `ESLE_Reporter`
Inheritance: `None`
File `ESLE_Reporter.H`

This is a pure virtual class, and in this sense an interface. However, it does not follow the conventions defined in Appendix A of [SLE-API].

This class provides a base for implementing the methods of the ISLE_Reporter interface that are used by the API for passing of log messages and notifications to the application. The application implements the `LogRecord` and Notify methods by specialising this class.

The `Create_ISLE_Reporter` method is used to obtain a MT-safe Reporter interface for configuration of SLE API and DCW. The interface is implemented in a MT-safe way using the methods of the given object.

Synopsis

```
class ESLE_Reporter
{
public:
  virtual
    ~ESLE_Reporter();
  virtual void
    LogRecord( SLE_Component component,
               const ISLE_SII* psii,
               SLE_LogMessageType type,
               const char* message ) = 0;
  virtual void
    Notify( SLE_Alarm alarm,
            SLE_Component component,
            const ISLE_SII* psii,
            const char* text ) = 0;
  static ISLE_Reporter*
    Create_ISLE_Reporter( ESLE_Reporter* preporter );
```

```
};
```

Methods

```
void LogRecord( SLE_Component component,
                const ISLE_SII* psii,
                SLE_LogMessageType type,
                const char* message );
```

Enters a message into the system log. See [SLE-API] Section 6.9.2. Implemented by the client application.

```
void Notify( SLE_Alarm alarm,
             SLE_Component component,
             const ISLE_SII* psii,
             const char* text );
```

Notifies the application of a specific event. See [SLE-API] Section 6.9.2.

Implemented by the client application.

```
ISLE_Reporter* Create_ISLE_Reporter(ESLE_Reporter* preporter);
```

Creates and returns a pointer to a MT-safe `ISLE_Reporter` interface suitable for configuration of SLE API. The interface is implemented in a MT-safe way using the methods of the given object. The object passed with the argument is assumed to be owned by the MT-safe Reporter and will be eventually deleted by that object.

Arguments

`preporter`	pointer to `ELSE_Reporter` based object.

5.2.5 Class Trace

Name `ESLE_Trace`
Inheritance: `None`
File `ELSE_Trace.H`

This is a pure virtual class, and in this sense an interface. However, it does not follow the conventions defined in Appendix A of [SLE-API].

This abstract class provides a base for implementing the method of the ISLE_Trace interface that is used by the API for passing trace messages to the application.

The application implements the `TraceRecord` method by specialising this class.

The `Create_ISLE_Trace` method is used to obtain a MT-safe Trace interface that is passed to the Trace Control interface of API components when tracing is required.

Synopsis

```
class ESLE_Trace
{
public:
  virtual
    ~ESLE_Trace();
  virtual void
    TraceRecord( SLE_Component component,
                 const ISLE_SII* psii,
                 const char* text ) = 0;
  static ISLE_Trace*
    Create_ISLE_Trace(ELSE_Trace* ptrace);
};
```

Methods

```
void TraceRecord( SLE_Component component,
                  const ISLE_SII* psii,
                  const char* text );
```

Processes a trace record. See [SLE-API] Section 6.9.3. Implemented by the client application.

```
ISLE_Trace* Create_ISLE_Trace(ESLE_Trace* ptrace);
```

Creates and returns a pointer to a MT-safe `ISLE_Trace` interface suitable for use by the SLE API. The interface is implemented in a MT-safe way using the method of the given object. The object passed with the argument is assumed to be owned by the MT-safe Trace object and will be eventually deleted by that object.

Arguments	
`ptrace`	pointer to an `ELSE_Trace` based object that implements tracing.
Return value	pointer to a Trace interface for output of trace records.

5.2.6 Class Time Source

Name `ESLE_TimeSource`
Inheritance: `None`
File `ELSE_TimeSource.H`

This is a pure virtual class, and in this sense an interface. However, it does not follow the conventions defined in Appendix A of [SLE-API].

This abstract class provides a base for implementing the method of the ISLE_TimeSource interface that is used by the API to obtain the current time.

The application implements the `Get_CurrentTime` method by specialising this class.

The `Create_ISLE_TimeSource` method is used to obtain a MT-safe Time Source interface that is passed to the function creating the utility factory.

Synopsis

```
class ESLE_TimeSource
{
public:
  virtual
    ~ESLE_TimeSource();
  virtual SLE_Octet*
    Get_CurrentTime() const = 0;
  static ISLE_TimeSource*
    Create_ISLE_TimeSource(ELSE_TimeSource* ptimeSource);
};
```

Methods

```
virtual SLE_Octet* Get_CurrentTime() const = 0;
```

Returns current time in CCSDS CDS format.

```
ISLE_TimeSource*
  Create_ISLE_TimeSource(ESLE_TimeSource* ptimeSource);
```

Creates and returns a pointer to a MT-safe interface `ISLE_TimeSource` suitable for use by the SLE API. The interface is implemented in a MT-safe way using the method of the given object. The object passed with the argument is assumed to be owned by the MT-safe Time Source object and will be eventually deleted by that object.

Arguments

`ptimeSource`	pointer to an `ESLE_TimeSource` based object that implements access to the time source.

5.2.7 Class Communication Server

Name `ESLE_CommunicationServer`

Inheritance: `None`

File `ELSE_CommunicationServer.H`

This is a standard C++ class and not an interface following the conventions defined in [SLE-API], Appendix A.

This class provides methods to initialise and shutdown the SLE API communication server.

The interface is a singleton accessed via the class method `Get_ESLE_CommunicationServer`.

The application first calls `Initialise`, which instantiates and configures the SLE API components in the communication server. The `Initialise` function takes the time source and reporter interfaces as arguments; if not NULL, these interfaces are used for getting the current time and for default logging throughout the communicatiobn server.

After initialisation the communication server is ready to accept connect requests and the subsequent SLE operation invocations from the SLE user applications.

If the configuration needs to be updated on-line, the application calls `UpdateConfiguration()` supplying the new SLE API proxy configuration file path as argument. On success, the communication server will perform processing based on the new configuration file; existing associations are not touched during configuration change.

In case the application needs to switch on tracing, it calls the `StartTrace` function supplying the `ISLE_Trace` interface and the tracing level. The communication server forwards then all tracing information to the supplied interface. In order to stop tracing, the function `StopTrace` must be called.

Finally the application releases the ESA SLE API components using the `Shutdown` method.

After a `Shutdown`, it is assumed that the application process will terminate. Thus it is not possible to reinitialise and restart the communicatiuon server after a `Shutdown`.

Synopsis

```
#include "ISLE_TimeSource.h"
#include "ISLE_Trace.h"
#include "ISLE_Reporter.h"


class ESLE_CommunicationServer
{
public:
  static ESLE_CommunicationServer&
     Get_ESLE_Communication Server();
  HRESULT
     Initialise (ISLE_Reporter* preporter,
                 ISLE_TimeSource* ptimeSource,
                 const char* proxyConfigFilePath);
  HRESULT
     UpdateConfiguration (const char* proxyConfigFilePath);
  HRESULT
     StartTrace (ISLE_Trace* trace,
                 SLE_TraceLevel level);
```

```
  HRESULT
     StopTrace ();
  HRESULT
     Shutdown ();

};
```

Methods

`ESLE_CommunicationServer& Get_ESLE_CommunicationServer();`

Class method that returns the `ESLE_CommunicationServer` singleton object.

`HRESULT Initialise( ISLE_Reporter* preporter,`
`ISLE_TimeSource* ptimeSource`
`const char* proxyConfigFilePath);`

Creates and configures the Communication Server, ESA TCP/IP Proxy, ESA SLE Utilities and ESA SLE Operations components.

Arguments:

`preporter`	pointer to a MT-safe Reporter interface for passing of log messages and notifications from the ESA SLE API communication server. If the argument is zero, no default log messages are supplied.
`ptimeSource`	pointer to time source interface to be used to obtain the current time. If the argument is not supplied or zero, the current time is obtained from the system time.
`ProxyConfigFilePath`	path of the Proxy configuration file

Result Codes:

`S_OK`	initialisation & configuration completed without errors
`SLE_E_NOFILE`	configuration data file not found
`SLE_E_CONFIG`	error or inconsistency in configuration data
`SLE_E_STATE`	already initialised
`E_FAIL`	failure due to unspecified error

`HRESULT UpdateConfiguration( const char* proxyConfigFilePath);`

Reads the configuration based on the proxy configuration file; on success, the communication server will perform processing based on the new configuration file; existing associations are not touched during configuration change.

Arguments:

`ProxyConfigFilePath`	path of the Proxy configuration file

Result Codes:

`S_OK`	configuration update completed without errors
`SLE_E_NOFILE`	configuration data file not found
`SLE_E_CONFIG`	error or inconsistency in configuration data
`SLE_E_STATE`	not yet initialised: `Initialise()` was not yet called
`E_FAIL`	failure due to unspecified error

`HRESULT StartTrace( ISLE_Trace* trace, SLE_TraceLevel level);`

Starts tracing in the SLE communication server based on the supplied trace level.

Arguments:

`trace`	the tracing interface which takes the trace records
`level`	the tracing level

Result Codes:

S_OK	tracing starts
E_FAIL	failure due to unspecified error

HRESULT StopTrace();

Stops tracing in the SLE communication server.

Result Codes:

S_OK	tracing stopped
E_FAIL	failure due to unspecified error

HRESULT ShutDown();

Stops processing in the SLE communication server.

Result Codes:

S_OK	shutdown initiated
E_FAIL	failure due to unspecified error

5.3 Default Logger Interface

5.3.1 Overview

The default logger provides access to log records and notifications generated by the TCP communications server process, which are not related to a specific service instance. A single object in a single process on a system can connect to the communications server at a time. If no process is connected, log messages are discarded. The interface can additionally be used to control tracing by the communications server process.

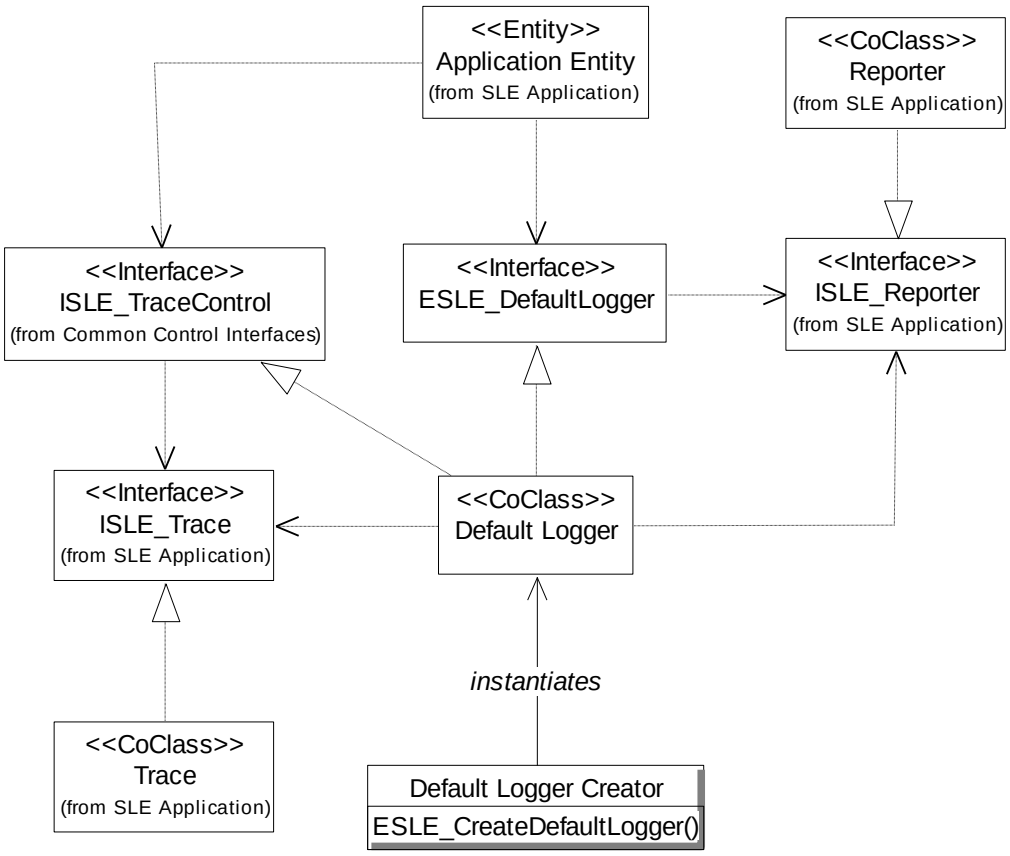

Figure 5-3 Default Logger

Figure 5-3 shows the classes supporting this interface. The creator function `ESLE_CreateDefaultLogger()` creates a new object and returns a pointer to the interface `ESLE_DefaultLogger`. The interface `ISLE_Reporter` must be passed to the default logger to forward log messages and notifications from the communication server process. Logging is started and stopped by the methods `Connect()` and `Disconnect()` defined by the default logger interface. To control traces, the interface `ISLE_TraceControl` can be obtained using the inherited method `QueryInterface` on `ESLE_DefaultLogger`.

5.3.2 Default Logger Creator Function

File DFL.H

The DFL package includes a function to create an instance and obtain a pointer to the default logger. The signature of this function is defined as:

```
extern "C" HRESULT ESLE_CreateDefaultLogger(const GUID& dflIid,
                                            void** pidfl);
```

Creates the default logger and returns a pointer to the specified interface. The function creates a single object in a process. If the object already exists it returns a pointer to that object.

Result Codes:

`S_OK`	the object has been instantiated
`E_NOINTERFACE`	the interface specified is not supported
`E_FAIL`	failure due to unspecified error

5.3.3 Default Logger Interface

Name `ESLE_DefaultLogger`
GUID `{3646D680-5197-11d3-9F15-00104B4F22C0}`
Inheritance: `IUnknown`
File `ESLE_DefaultLogger.H`

This interface provides the means to receive log messages and notifications from the communications server process and to control tracing by the communications server process via the interface `ISLE_TraceControl`, available via the inherited method `QueryInterface`.

Synopsis

```
#include <SLE_SCM.H>
#include <SLE_APITypes.h>

interface ISLE_Reporter;

#define IID_ESLE_DefaultLogger_DEF { 0x3646d680, 0x5197, 0x11d3, \
          { 0x9f, 0x15, 0x0, 0x10, 0x4b, 0x4f, 0x22, 0xc0 } }

interface ESLE_DefaultLogger : IUnknown
{
  virtual void
    Set_Reporter(ISLE_Reporter* preporter) = 0;
  virtual HRESULT
    Connect(const char* ipcAddress) = 0;
  Virtual HRESULT
    Disconnect() = 0;
};
```

Methods

`void Set_Reporter(ISLE_Reporter* preporter);`

Passes the reporter interface to which log messages and notifications shall be forwarded.

Arguments

`preporter`	pointer to the reporter interface

`HRESULT Connect(const char* ipcAddress);`

Connects to communications server process.

Arguments

`ipcAddress`	The IPC address for default logging and tracing. This address must be the same as the one entered in the configuration file for the communications server.

Result Codes:

`S_OK`	the connection has been established
`SLE_E_STATE`	the object is already connected
`SLE_E_CONFIG`	no reporter interface available
`E_FAIL`	unable to connect to the communications server

`HRESULT Disconnect();`

Disconnects from the communications server process. This method must be called before `Release()` is called.

Result Codes:

`S_OK`	the connection has been released
`SLE_E_STATE`	the object is not connected

5.4 C Language Interface Concept

5.4.1 Introduction

The C Language SLE API provides a C language interface to the (C++) ESA SLE API. For each method of each supported interface[5] it provides a C function which maps onto an invocation of the method. The reference of the target interface is passed as the first parameter of each function.

The ESA SLE API supports the C language interface for the following services:

- Return All Frames
- Return Channel Frames
- Forward Communication Link Transmission Unit

The ESA SLE API does not support a C language interface for the services ROCF and FSP.

The following sections provide a general explanation of how the C-API maps onto the ESA SLE API.

5.4.2 Interface Name Acronyms

Each interface is assigned an acronym that is used as a prefix when forming names of functions and types related to the interface in the C-API. This causes C++ class-scoped names to be unique in C's global namespace. Acronyms are used because using the interface name leads to very long function names. Table 5-2 shows the interfaces supported by the C-API with their acronyms.

5.4.3 Object reference types

An opaque type is provided to represent a reference to an object interface for each interface in the C-API.

Post-fixing the suffix _Obj to the interface acronym forms the name of the opaque type. An additional opaque type is provided for constant interface pointers. Post-fixing the suffix _CObj to the interface acronym forms the name of this opaque type.

The opaque type is defined as a pointer to a structure named after the interface (note the C++ interface declaration is equivalent to a C struct). The definition of the structure is not visible in the C-API so the type is incomplete. This means that variables of this type can be instantiated and assigned to, with compile-time type checking, but the pointer value cannot be de-referenced.

E.g. for an `ISLE_Operation` (acronym `ISLE_OP`) the following type definition is generated:

```
typedef struct ISLE_Operation* ISLE_OP_Obj;
```

For further examples see section 5.4.11.

5.4.4 Pass By Reference Parameters

In C++ parameters can be explicitly passed by reference, using the **&** operator. The C-API implements pass-by-reference is as pass-by-address, thus ***** replaces **&** in function declarations.

5.4.5 Methods

A C function is provided for each method and each inherited method of each interface in the C-API.

[5] For the purpose of mapping to C, the term *interface* is interchangeable with the C++ term *class* or *struct*. It refers to classes of the API Builder package as well as the COM style DCW and SLE API interfaces. Likewise the term *method* is interchangeable with the term C++ member function.

The first parameter of the C function is always the reference to the object to which the function is to be applied.

The return type and other parameters are the same as those of the method except that pointers and references to objects are replaced with their corresponding object reference types.

The C function name is formed from the method name by prefixing it with the interface acronym, separated by an underscore. E.g. `QueryInterface` of interface `IUnknown` becomes `ISLE_IU_QueryInterface`. For further examples see section 5.4.11.

Package	**Interface Name**	**C-API acronym**
API Builder	ESLE_Builder	ESLE_B
SLE Application	ISLE_Reporter	ISLE_REP
	ISLE_Trace	ISLE_TR
	ISLE_TimeSource	ISLE_TS
Common Control Interfaces	ISLE_TraceControl	ISLE_TCT
Down Call Wrapper	IDCW_EventQueue	IDCW_EVQ
	IDCW_Admin	IDCW_AD
	IDCW_SIFactory	IDCW_SIF
SLE Operations	ISLE_Bind	ISLE_BN
	ISLE_ConfirmedOperation	ISLE_COP
	ISLE_Operation	ISLE_OP
	ISLE_OperationFactory	ISLE_OPF
	ISLE_PeerAbort	ISLE_PAB
	ISLE_ScheduleStatusReport	ISLE_SSR
	ISLE_Stop	ISLE_STP
	ISLE_Unbind	ISLE_UBN
API Service Element	ISLE_SIAdmin	ISLE_SIA
	ISLE_SEAdmin	ISLE_SEA
	ISLE_SIOpFactory	ISLE_SIF
	ISLE_ServiceInitiate	ISLE_SIN
API Proxy	ISLE_ProxyAdmin	ISLE_PA
SLE-SCM	IUnknown	ISLE_IU
SLE Utilities	ISLE_SII	ISLE_SII
	ISLE_Credentials	ISLE_CRD
	ISLE_SecAttributes	ISLE_SA
	ISLE_Time	ISLE_TM
	ISLE_UtilFactory	ISLE_UF
RAF, RCF and CLTU replace <SRV> in the following entries:		
Service Supplement <SRV>	I<SRV>_SIAdmin	I<SRV>_SIA
	I<SRV>_SIUpdate	I<SRV>_SIU
	I<SRV>_Start	I<SRV>_ST

Package	Interface Name	C-API acronym
	I<SRV>_TransferData	I<SRV>_TD
	I<SRV>_SyncNotify	I<SRV>_SN
	I<SRV>_AsyncNotify	I<SRV>_ASN
	I<SRV>_StatusReport	I<SRV>_SR
	I<SRV>_GetParameters	I<SRV>_GP
	I<SRV>_ThrowEvent	I<SRV>_TEV
	I<SRV>_InvokeDirective	I<SRV>_ID

Table 5-2 Interface Acronyms

5.4.6 Overloaded Operators

In C++ it is possible to implement special versions of the C/C++ operators for a particular class. This is known as operator overloading.

C-API functions corresponding to overloaded operators are formed by replacing the C++ operator with a function name-stem, as specified in Table 5-3.

C++ Operator	C function name stem
==	Equal
!=	NotEqual
<	LessThan
>	GrtrThan
<=	LessThanEqual
>=	GrtrThanEqual
+	Add
-	Subtract

Table 5-3 Overloaded Operator Names

E.g.
```
bool IID::operator==(const IID&) const;
```
maps to
```
bool IID_Equal(IID_Obj, IID_Obj);
```

5.4.7 Static Member Functions

The C-API provides a wrapper function for a static member function following the same pattern as an interface method, except that there is no interface object parameter.

E.g. the C-API provides
```
ELSE_Builder_Obj ELSE_Get_ELSE_Builder();
```
for
```
ELSE_Builder& ELSE_Builder::Get_ELSE_Builder();
```

5.4.8 Up-Call Functions

The C-API provides up-call (call-back) functions that are following exactly the same pattern as for the down-call functions. These functions are:

```
void ISLE_REP_LogRecord( ... ),      declared in ISLE_Reporter_c.h
void ISLE_REP_Notify( ... ),         declared in ISLE_Reporter_c.h
ISLE_REP_AddRef(ULONG val),          declared in ISLE_Reporter_c.h
ISLE_REP_Release(ULONG val)),        declared in ISLE_Reporter_c.h

void ISLE_TR_TraceRecord( ... ),     declared in ISLE_Trace_c.h
ISLE_TR_AddRef(ULONG val),           declared in ISLE_Trace_c.h
ISLE_TR_Release(ULONG val)),         declared in ISLE_Trace_c.h

void ISLE_TS_GetCurrentTime(),       declared in ISLE_TimeSource_c.h
ISLE_TS_AddRef(ULONG val),           declared in ISLE_TimeSource_c.h
ISLE_TS_Release(ULONG val)),         declared in ISLE_TimeSource_c.h
```

These C functions have to be implemented by the application that uses the C-API, i.e. the C-body of these functions has to be provided by the application programmer. The functions `<>_AddRef()` and `<>_Release()` are called whenever the API adds/releases a reference to the object. When the argument '`val`' in the functions `<>_Release()` is zero, the client can release any resources allocated for reporting, tracing or for the time source.

The C-API also provides internal C++ to C wrapper classes, which call the functions listed above. In order to create objects of these wrapper classes, the programmer has to call the following object creation functions for the reporter, the trace and the time source objects (see Appendix C for the signature):

```
EE_CAPI_ACC_EE_CAPI_CreateTrace(const GUID* iid,
                                EE_BOOL addMTSWrapper, void** ppv)

EE_CAPI_ACC_EE_CAPI_CreateReporter(const GUID* iid,
                                EE_BOOL addMTSWrapper, void** ppv)

EE_CAPI_ACC_EE_CAPI_CreateTimeSource(const GUID* iid,
                                EE_BOOL addMTSWrapper, void** ppv)
```

The parameters of the creator functions are:

`iid`	the GUID of the up-call interface defined in reference [SLE-API]
`AddMTSWrapper`	if set to ‘true’ a multi-thread safe wrapper is added
`ppv`	the pointer to the created object

The so created objects for the reporter and the time source are foreseen to be passed to the API-Builder (see initialise method of builder: `ELSE_B_Initialise`), and the created object for the trace shall be passed when starting the trace (`ISLE_TCT_StartTrace()`).

The methods of the objects will be invoked by the API and then will call the C functions provided by the application.

This mechanism makes sure that the API calls the functions implemented by the C-client for logging, tracing and for obtaining the time source.

Example:

```
ISLE_REP_Obj p_isleReporter=NULL;
GUID tmpgRep = IID_ISLE_Reporter_DEF;
ISLE_TS_Obj p_isleTimeSource=NULL;
GUID tmpgTimes = IID_ISLE_TimeSource_DEF;
```

```
ESLE_B_Obj The_builder = ESLE_B_Get_ESLE_Builder();

/* create the MT Safe Reporter */
EE_CAPI_ACC_EE_CAPI_CreateTrace(&tmpgRep, true, (void**)&p_isleReporter);

/* create the MT Safe TimeSource */
EE_CAPI_ACC_EE_CAPI_CreateTimeSource(&tmpgTimes, true,
                                     (void**)&p_isleTimeSource);

/* initialise the builder */
if ((res = ESLE_B_Initialise(The_builder,"./SEConfigFile",
            "./ProxyConfigFile", p_isleReporter,
            sleBR_initiator,p_isleTimeSource)) == S_OK) {
}
```

5.4.9 Memory Management

5.4.9.1 Interface Object reference

Interface object references may only be freed by the client application using the method `Release()`. The use of `free()` on an object reference leads to undefined behaviour.

5.4.9.2 String Pointers and Arrays

C strings are returned from the C-API to the client application as the type `char*`. Byte arrays are passed as `SLE_Octet*` which maps to `unsigned char*`. The C-API provides special functions for these and any other non-object pointers that are allocated on the heap. The provided function should be used because the SLE-API might use different heap memory management than that used in C. The use of `free()` on non-object pointers can lead to undefined behaviour.

The function for freeing C strings returned from the C-API is:

```
void ESLE_String_Free(char*);
```

5.4.9.3 C Arrays and Structures

In order to release memory for C arrays and structures that are not ASCII strings, a separate function is available. To release memory allocated by the API, the following function has to be called:

```
void ESLE_Memory_Free(void*);
```

5.4.10 C-API Files

A C header file is provided for each interface. It `#include`'s all the header files it requires; i.e. the standard SLE API types header files. It defines the interface object reference type and declares the C functions corresponding to the interface. It also defines object reference types for the interfaces that its methods use.

A C++ wrapper implementation file is provided for each interface. It `#include`'s both the C and C++ header files for the interface and defines the functions that map from the C function call to the C++ method call.

The filenames are formed from the interface name with the suffix **_c**, e.g. for `ISLE_Operation` we have `ISLE_Operation_c.h, ISLE_Operation_c.cpp`.

5.4.11 Example

The example presented in Figure 5-4 and Figure 5-5 demonstrates the way the C-API is formed. The interface `Gen` is a base interface from which `Spec` inherits. The C interface acronyms are the same as the interface names. Note that C functions are formed for all `Spec`'s methods, plus those inherited from `Gen`.

5.4.12 Constraints

The following C++ concepts are not used in the SLE API so are not supported as concepts of the C-API:

- Pass-by-value of objects
- Return-by-value of objects
- Overloaded functions
- Direct access to C++ data members
- Namespaces
- Exceptions

```
Gen.H

#include <SLE_APITypes.h>

interface Aclass;
interface Gen
{
   virtual Aclass* funcx(const char*) = 0;
   virtual int funcy(const Aclass*) const = 0;
};

Spec.H

#include <SLE_APITypes.h>
#include <Gen.H>

interface Spec : public Gen
{
   virtual char* funcz(Aclass*) = 0;
};
```

Figure 5-4 C++ Interface declaration files

Gen_c.h

```
#include <SLE_APITypes.h>

typedef struct Aclass* Aclass_Obj;
typedef const struct Aclass* Aclass_CObj;
typedef struct Gen* Gen_Obj;
typedef const struct Gen* Gen_CObj;

Aclass_Obj   Gen_funcx(Gen_Obj, const char*);
int          Gen_funcy(Gen_CObj, Aclass_CObj);
```

Spec_c.h

```
#include <SLE_APITypes.h>

typedef struct Aclass* Aclass_Obj;
typedef const struct Aclass* Aclass_CObj;
typedef struct Gen* Gen_Obj;
typedef struct Spec* Spec_Obj;
typedef const struct Spec* Spec_CObj;

Aclass_Obj   Spec_funcx(Spec_Obj, const char*);
int          Spec_funcy(Spec_CObj, Aclass_CObj);
char*        Spec_funcz(Spec_Obj, Aclass_Obj);
```

Figure 5-5: C Interface declaration files

6 Software Development and Integration

6.1 Build Information for Solaris

6.1.1 Compiling a SLE Application

ESA SLE API client code must be compiled using an include directive with the path of the SLE API installation containing the SLE API header files. The compiler directive is

```
-I<sleapi-include-path>
```

Depending on the C++ compiler used, it may be required to use the pre-processor macro "_NOBOOL_ ", which defines the standard C++ type `bool` if that is not yet supported by the compiler. The following shows how the pre-processor macro is passed to the compiler:

```
-D_NOBOOL_
```

In addition the following compiler macro is needed on the Solaris in order to specify the event mechanism used (see [SLE-API], section 6.3.2):

```
-D_SLE_EH_FILE_DESCRIPTOR_
```

The following compiler macro on Solaris is needed for operating system specific compilation:

```
-D_EE_OS_SOLARIS_
```

6.1.2 Linking a SLE Application

To link the ESA SLE API client application program, a set of API shared library names must be passed to the linker. The following shared libraries must be used for linking, where the extension '`<n>`' denotes the version of the shared library:

`libsleapi.so.<n>`	SLE API service element and proxy.
`libsleext.so.<n>`	ESA SLE API extensions (see section 5), only needed if the application uses the API extensions.
`libsleinf.so.<n>`	Infrastructure objects needed in all components of the ESA SLE API.
`libslec.so.<n>`	SLE C API. This library is only needed if the C programming language interface is used rather than the C++ interface of the API.

To use the operational (non-debug) API libraries, the application must be linked with the libraries, `libsleapi`, `libsleext`, `libsleinf`, and `libslec`.

The ESA SLE API libraries reference objects contained in the following libraries, which must also be added to the list of used libraries:

`libasnsdk_50_sol8`	The ASN.1 runtime library used for encoding/decoding for Solaris 8 (part of API delivery)
`libposix4`	Posix.1b real-time extensions library (e.g. semaphores).
`libsocket`	Socket library.
`libpthread`	Posix thread library.

For the use of the operational API libraries, the following linker option must be set:

```
-L<sleapi-library-path> -lsleapi –lsleext –lsleinf \\
```

```
-lasnsdk_50_sol<8> –lposix4 –lsocket –lpthread
```

6.2 Build Information for Linux

6.2.1 Compiling a SLE Application

ESA SLE API client code must be compiled using an include directive with the path of the SLE API installation containing the SLE API header files. The compiler directive is

```
-I<sleapi-include-path>
```

Depending on the C++ compiler used, it may be required to use the pre-processor macro "_NOBOOL_ ", which defines the standard C++ type `bool` if that is not yet supported by the compiler. The following shows how the pre-processor macro is passed to the compiler:

```
-D_NOBOOL_
```

In addition the following compiler macro is needed on Linux in order to specify the event mechanism used (see [SLE-API], section 6.3.2):

```
-D_SLE_EH_FILE_DESCRIPTOR_
```

The following compiler macro on Linux is needed for operating system specific compilation:

```
-D_EE_OS_LINUX_
```

6.2.2 Linking a SLE Application

To link the ESA SLE API client application program, a set of API shared library names must be passed to the linker. The following shared libraries must be used for linking, where the extension '`<n>`' denotes the version of the shared library:

`libsleapi.so.<n>`	SLE API service element and proxy.
`libsleext.so.<n>`	ESA SLE API extensions (see section 5), only needed if the application uses the API extensions.
`libsleinf.so.<n>`	Infrastructure objects needed in all components of the ESA SLE API.
`libslec.so.<n>`	SLE C API. This library is only needed if the C programming language interface is used rather than the C++ interface of the API.

To use the operational (non-debug) API libraries, the application must be linked with the libraries, `libsleapi`, `libsleext`, `libsleinf`, and `libslec`.

The ESA SLE API libraries reference objects contained in the following libraries, which must also be added to the list of used libraries:

`libasnsdk_50_sles11`	The ASN.1 runtime library used for encoding/decoding for Linux SLES 11 32 bit (part of API delivery)
`libasnsdk_50_sles12_linux64bit_gcc4`	The ASN.1 runtime library used for encoding/decoding for Linux SLES 12 64 bit (part of API delivery)
`libasnsdk_50_linux64bit_gcc4`	The ASN.1 runtime library used for encoding/decoding for Linux SLES 11 64 bit (part of API delivery)
`libposix4`	Posix.1b real-time extensions library (e.g. semaphores).

`libsocket`	Socket library.
`libpthread`	Posix thread library.

For the use of the operational API libraries, the following linker option must be set:

```
-L<sleapi-library-path> -lsleapi –lsleext –lsleinf \\
-l asnsdk_50_<sles11 or linux64bit_gcc4 or sles12_linux64bit_gcc4> -l
–lposix4 –lsocket –lpthread
```

6.3 Build Information for MS Windows 2003

6.3.1 Compiling a SLE Application

ESA SLE API client code must be compiled using an include directive with the path of the SLE API installation containing the SLE API header files. The compiler directive is

```
/I <sleapi-include-path>
```

In addition the following compiler macro is needed on NT4.0 / Windows 2003 in order to specify the event mechanism used (see [SLE-API], section 6.3.2): the following pre-processor definition must be applied:

```
/D _SLE_EH_EVENT_OBJECT_
```

In order to guarantee correct inclusion of IUnknown header files, the following directive must be applied during compilation:

```
/D EE_OS_NT4
```

6.3.2 Linking a SLE Application

To link the ESA SLE API client application program, a set of API library names must be passed to the linker:

`APIUT.lib`	SLE API utilities.
`APIOP.lib`	SLE API operation objects.
`APISE.lib`	SLE API service element.
`APIPX.lib`	SLE API proxy.
`SLEEXT.lib`	ESA SLE API extensions (see section 5), only needed of the application uses the API extensions.
`SLEINF.lib`	Infrastructure objects needed in all components of the ESA SLE API.
`SLEC.lib`	SLE C API. This library is only needed if the C programming language interface is used rather than the C++ interface of the API.

To use the operational (non-debug) API libraries, the application must be linked with the libraries, `APIUT.lib`, `APIOP.lib`, `APISE.lib`, `APIPX.lib`, `SLEEXT.lib` and `SLEC.lib`.

The ESA SLE API libraries reference objects contained in the following libraries, which must also be added to the list of used libraries:

`ws2_32.lib`	The Win32 API Socket library

`msvcrt.lib`	The C library.

7 Configuration of the API

7.1 Introduction

Configuration parameters for the API components API Service Element and API Proxy must be specified in text files following the syntax described in section 7.4. If the API builder is used, the name of the files and their location must be passed as an argument to the `Initialise()` method defined in the administrative interface of the API builder.

The API Service Element and the API Proxy are individually configurable according to the role they play in the application (user or provider), therefore separate configuration files must be provided, which partially contain redundant information. Consistency of the information must be ensured when these files are prepared.

The ESA SLE API package defines four different configuration files for the following components:

- the API Service Element supporting a SLE user application;
- the API Service Element supporting a SLE provider application;
- the API Proxy supporting the initiator role (SLE user application);
- the API Proxy supporting the responder role (SLE provider application).

The configuration files are read when the API is configured and when the Communications Server and the Default Logger are started. If the contents of a configuration file is modified subsequently, this has no effect on the API.

7.2 Configuration of an API Service Element

7.2.1 Configuration of an API Service Element Supporting the User Role

The configuration file for an API Service Element supporting the user role contains the following parameters

Application Role

Must be set to “user”.

Port to Protocol Mapping

This is a list of pairs of peer responder port identifiers and protocol identifiers used to route a BIND invocation to the proxy supporting the specified protocol. A default protocol identifier can be defined, which will be used for port identifiers that are not contained in the list.

If an application uses a single type of proxy, entry of the default protocol identifier is sufficient.

The proxy in the ESA SLE API package is identified by the protocol ID “ISP1”.

Maximum Trace Length

Defines the maximum length of trace output.

7.2.2 Configuration of an API Service Element Supporting the Provider Role

The configuration file for an API Service Element supporting the provider role contains the following parameters

Application Role

Must be set to “provider”.

Port to Protocol Mapping

The SE must determine the protocol ID for a logical port, before it can register the port to a proxy.

This is a list of pairs of local port identifiers and protocol identifiers used to determine the proxy to which a local port can be registered. A default protocol identifier can be defined, which will be used for port identifiers that are not contained in the list.

If an application uses a single type of proxy, entry of the default protocol identifier is sufficient.

The proxy in the ESA SLE API package is identified by the protocol ID “ISP1”.

Reporting Frequency Range

Defines the acceptable reporting frequency that a user may request in the SCHEDULE-STATUS-REPORT operation. It is noted that this range is constrained to 2 to 600 seconds by the CCSDS Recommendations. The service element will accept a frequency that falls within the range defined in the configuration file.

Maximum reporting frequency – specified in seconds

Minimum reporting frequency – specified in seconds

Maximum Trace Length

Defines the maximum length of trace output.

7.3 Configuration of an API Proxy

7.3.1 Configuration of an Initiating Proxy in a SLE User Application

The configuration file for an API Proxy supporting the initiator role contains the following parameters

BIND Role

Must be set to “initiator”.

Local Application Identification

Defines the user name and password of the local application. The parameter has the following elements.

Authority Identifier

The authority identifier (user name) entered as a character string.

Password

The password entered as a sequence of hexadecimal digits.

SLE Service Type List

Lists the service types and versions that are supported by the application and all API components. Each entry contains the following parameters.

Service Type Abbreviation

The abbreviated name of the service type (e.g. RAF).

Version Number List

The list of versions supported by the application and all API components, starting with the highest version number.

List of Peer Applications

The list of SLE provider applications, where every peer application is defined by the following parameters

Authority Identifier

The authority identifier (user name) entered as a character string.

Authentication Mode

Must have one of the values:
- “none” – no PDUs shall be authenticated
- “bind” – only the BIND PDUs shall be authenticated
- “all” – all PDUs shall be authenticated.

Password

The password entered as a sequence of hexadecimal digits. This entry is not needed when the authentication mode is set to “none”.

List of Foreign Responder Ports

The list of responder ports to which the proxy shall connect and their mapping to TCP sockets. The following parameters describe a responder port.

Logical Port Identifier

The logical port identifier as assigned by service management, entered as a character string.

Socket List

Lists the TCP sockets to which the port identifier maps. Multiple sockets must only be entered when the special connection establishment procedure defined in reference [TCP-PROXY] shall be applied. Every socket is defined by the following parameters.

IP Address

The IP address of the host can be entered in one of the following forms
- an IP version 4 address in dot notation;
- a fully qualified DNS host name.

If the host name is used, the operating system must be configured to deliver the IP address either from local definitions or using a DNS server.

TCP Port Number

The port number in decimal format

TCP buffer size

TCP Transmit Buffer Size

The TCP transmit buffer size in bytes, default value is 0. If set to 0, the buffer size reserved from the operating system is effective.

TCP Receive Buffer Size

The TCP send buffer size in bytes, default value is 0. If set to 0, the buffer size reserved from the operating system is effective.

Heartbeat Interval

Defines the heartbeat interval defined in reference [TCP-PROXY] in seconds. A value of zero indicates that the heartbeat mechanism shall not be used.

Dead-Factor

Defines the dead-factor defined in reference [TCP-PROXY].

Output Queue Length

Defines the maximum number of PDUs that the proxy shall queue for transmission.

Maximum Authentication Delay

Defines the maximum delay in seconds between the time credentials have been generated and the time authentication of the credentials is performed.

Maximum Trace Length

Defines the maximum length of trace output.

7.3.2 Configuration of a Responding Proxy in a SLE Provider Application

The configuration file for an API Proxy supporting the responder role contains the following parameters

Bind Role

Must be set to “responder”.

Local Application Identification

Defines the user name and password of the local application. The parameter has the following elements.

Authority Identifier

The authority identifier (user name) entered as a character string.

Password

The password entered as a sequence of hexadecimal digits.

SLE Service Type List

Lists the service types and versions that are supported by the application and all API components. Each entry contains the following parameters.

Service Type Abbreviation

The abbreviated name of the service type (e.g. RAF).

Version Number List

The list of versions supported by the application and all API components, starting with the highest version number.

List of Peer Applications

The list of SLE user applications that may use the services provided, where every peer application is defined by the following parameters

Authority Identifier

The authority identifier (user name) entered as a character string.

Authentication Mode

Must have one of the values:

- “none” no PDUs shall be authenticated
- “bind” only the BIND PDUs shall be authenticated
- “all” all PDUs shall be authenticated.

Password

The password entered as a sequence of hexadecimal digits. This entry is not needed when the authentication mode is set to “none”.

List of Local Responder Ports

Lists one or more responder ports on which the proxy shall accept BIND invocations.

Logical Port Identifier

The logical port identifier as assigned by service management, entered as a character string.

Socket

The TCP socket, to which the port identifier maps, defined by the following parameters. More than one logical port identifier can map to the same TCP socket.

IP Address

The IP address of the host can be entered in one of the following forms

- an IP version 4 address in dot notation;
- “*” (indicating 'any') – TCP connections shall be accepted on any IP address defined on the local host

TCP Port Number

The port number in decimal format

TCP buffer size

TCP Transmit Buffer Size

The TCP transmit buffer size in bytes, default value is 0. If set to 0, the buffer size reserved from the operating system is effective.

TCP Receive Buffer Size

The TCP send buffer size in bytes, default value is 0. If set to 0, the buffer size reserved from the operating system is effective.

Output Queue Length

Defines the maximum number of PDUs that the proxy shall queue for transmission.

Maximum Authentication Delay

Defines the maximum delay in seconds between the time credentials have been generated and the time authentication of the credentials is performed.

TML Parameters

The transport mapping layer defined in reference [TCP-PROXY] requires the following configuration parameters.

TML Start-up Timer

Defines the maximum delay in seconds between establishment of a TCP connection and arrival of the first message.

Non Use of Heartbeat Mechanism

Defines whether or not a value of zero shall be accepted for the heartbeat interval indicating that the heartbeat mechanism shall not be used.

Heartbeat Interval Range

Defines the minimum and maximum values of the heartbeat interval that the proxy shall accept.

Dead-Factor Range

Defines the minimum and maximum values of the dead-factor that the proxy shall accept.

IPC Configuration

Communication between the proxy and the communications server process is controlled by the following parameters.

Communications Service IPC Address

Solaris platform: Defines a named IPC channel by which the proxy component can connect to the communications server process (In UNIX this is expected to be a named pipe).

Linux platform: Defines a local TCP port by which the proxy component can connect to the communications server process.

Default Logging IPC Address

Solaris platform: Defines a named IPC channel by which the default logger can connect to the communications server process (In UNIX this is expected to be a named pipe).

Linux platform: Defines a local TCP port by which the default logger process can connect to the communications server process.

Maximum Trace Length

Defines the maximum length of trace output.

7.3.3 Configuration of the TCP Communications Server Process

The TCP communications server process uses the same configuration file as a responding proxy (see section 7.3.2) and ignores those entries it does not use. The file passed to the communications server process must contain the peer applications and the responder port identifiers defined for all SLE applications on the host.

When all SLE application processes running on one host use the same authority identifier, the same file can be used for all proxies and for the communications server process. If different files are used, they must be consistent.

If SLE applications running on the same host use different authority identifiers, a default identifier must be defined for communications server process (in the parameter Authority Identifier of the configuration file used by the communications server process). This identifier will be used only when a BIND invocation with an unknown service instance identifier must be rejected.

7.4 Syntax of the Configuration Files

This section describes the syntax of the configuration files using an adapted BNF notation. The elements of the notation are:

`<x>`	an item that is further expanded
`x := y`	left hand item x is formed by the right hand production y
`[x]`	optional item (zero or one)
`[x]*`	zero to N
`[x]+`	one to N
`x | y`	x OR y
`(x)`	complete production x treated as a single item
`'x'`	literal
`--`	comment in the syntax description

The following terminals are used without further expansion

`identifier`	a character string without white space
`dec-number`	a decimal number, optionally followed by a range indication in brackets
`hex-string`	a string of hexadecimal digits
`string`	a character string
`IP-address`	a version 4 IP address in dot notation
`DNS-name`	a valid DNS host name
`NL`	end of a line

General rules, that are not specifically expressed by the notation are:

- Keywords are case insensitive (capital letters are used in this specification)
- Identifiers for the same type of items at the same nesting level must be unique
- White-space may but need not be used to separate tokens
- The sequence of the top level items is not significant

Comments can be entered anywhere in the files using the hash (`'#'`) character. All characters following `'#'` up to the end of the line are discarded.

Service Element Configuration File - User Side

```
<se-user-config-file> :=
   'APPLICATION_ROLE' '=' 'USER' NL
   <application-identifier>
   <port-protocol-mapping>
   <trace-length>
```

Service Element Configuration File - Provider Side

```
<se-provider-config-file> :=
   'APPLICATION_ROLE' '=' 'PROVIDER' NL
   <port-protocol-mapping>
   <reporting-frequency-range>
   <trace-length>
```

Proxy Configuration File - User Side

```
<proxy-user-config-file> :=
   'PROXY_ROLE' '=' 'INITIATOR' NL
   <local-application-id>
   <service-type-list>
   <peer-application-list>
   <foreign-responder-port-list>
   <transmit-queue-length>
```

```
   <authentication-delay>
   <trace-length>
```

Proxy Configuration File - Provider Side

```
<proxy-provider-config-file> :=
   'PROXY_ROLE' '=' 'RESPONDER' NL
   <local-application-id>
   <service-type-list>
   <peer-application-list>
   <local-responder-port-list>
   <transmit-queue-length>
   <authentication-delay>
   <tml-parameters>
   <ipc-config>
   <trace-length>
```

Common Syntax Elements

```
<address-list> :=
   <ip-address-list> | <host-name-list>

<addr-string> := IP-address | '*'
   -- '*' only allowed in local addresses

<application-identifier> :=
   'APPLICATION_ID' '=' identifier NL

<authentication-delay> :=
   'AUTHENTICATION_DELAY' '=' dec-number( ≥1 ) NL

<authentication-mode> := 'NONE' | 'BIND' | 'ALL'

<cs-address> :=
   'CS_ADDRESS' '=' identifier NL

<dead-factor-range> :=
   'MIN_DEADFACTOR' '=' dec-number( >0 ) NL
   'MAX_DEADFACTOR' '=' dec-number( >0 ) NL

<default-reporting-address> :=
   'DEFAULT_REPORTING_ADDRESS' '=' identifier NL

<f-port> :=
   'PORT_NAME' '=' identifier NL
   'PORT_HEARTBEAT_TIMER' '=' dec-number (( ≥0 ) NL
   'PORT_DEAD_FACTOR' '=' dec-number (( ≥1 ) NL
   <address-list>
...<tcp-buffer-sizes>
<foreign responder-port-list> :=
   'FOREIGN_LOGICAL_PORTS '=' '{' NL [<f-port>]+ '}' NL

<heartbeat-timer-range> :=
   'MIN_HEARTBEAT' '=' dec-number( >0 ) NL
   'MAX_HEARTBEAT' '=' dec_number( >0 ) NL

<host-name-and-port> := DNS-name ':' dec-number NL

<host-name-list> :=
```

```
   'HOST_NAME' '=' '{' NL [<host-name-and-port>]+ '}' NL

<ip-addr-and-port> :=
   <addr-string> ':' dec-number NL

<ip-address-list> :=
   'IP_ADDRESS' '=' '{' NL [<ip-addr-and-port>]+ '}' NL

<ipc-config> :=
   <cs-address>
   [<use-nagle>]
   <default-reporting-address>

<local-application-id> :=
   'LOCAL_ID' '=' identifier NL
   'LOCAL_PASSWORD' '=' hex-string NL

<local-responder-port-list> :=
   'LOCAL_LOGICAL_PORTS' '=' '{' NL [<l-port>]+ '}' NL

<l-port> :=
   'PORT_NAME' '=' identifier NL
   <ip-address-list>
...<tcp-buffer-sizes>

<non-use-of-heartbeat-allowed> :=
   'NON_USEHEARTBEAT' '=' ('TRUE' | 'FALSE') NL

<peer-appliction-list> :=
   'REMOTE_PEERS' '=' '{' NL [<peer-spec>]+ '}' NL

<peer-spec> :=
   'ID' '=' identifier NL
   'PASSWORD' '=' hex-string NL
   'AUTHENTICATION_MODE' '=' <authentication-mode> NL

<port-protocol-mapping> :=
   'PORTLIST' '=' '{' NL [<pp-mapping>]*
                         ['DEFAULT' <protocol-id> NL] '}' NL
   -- if the default protocol is not specified, at least one
   -- mapping must be entered

<port-identifier> := identifier

<pp-mapping> :=
   <port-identifier> '=' <protocol-id> NL

<protocol-id> := identifier  -- only 'ISP1' currently allowed




<reporting-frequency-range> :=
   'MIN_REPORTINGCYCLE' '='dec-number( >0 ) NL
   'MAX_REPORTINGCYCLE' '='dec-number( >0 ) NL

<service-id> :=
   'CLTU' | 'FSP' |'RAF' | 'RCF' | 'ROCF'
```

```
<service-type> :=
   'SRV_ID' '=' <service-id> NL
   'SRV_VERSION' '=' '{' NL [<service-version>]+ '}' NL

<service-type-list> :=
   'SERVER_TYPES '=' '{' NL [<service-type>]+ '}' NL

<service-version> := dec-number( ≥1 ) NL -- currently only '1'

<startup-timer> :=
   'STARTUP_TIMER' '=' dec-number( >0 ) NL

<tcp-buffer-sizes> :=
...[<tcp-receive-buffer-size>]
...[<tcp-transmit-buffer-size>]

<tcp-receive-buffer-size> :=
   'TCP_RECV_BUFFER_SIZE' '=' dec-number( ≥0 ) NL

<tcp-transmit-buffer-size> :=
   'TCP_XMIT_BUFFER_SIZE' '=' dec-number( ≥0 ) NL

<tml-parameters> :=
   <startup-timer>
   <heartbeat-timer-range>
   <dead-factor-range>
   <non-use-of-heartbeat-allowed>

<trace-length> :=
   'MAX_TRACE_LENGTH' '=' dec-number( ≥0 )

<transmit-queue-length> :=
   'TRANSMIT_QUEUE_SIZE' '=' dec-number( ≥1 ) NL

<use-nagle> :=
   'USE_NAGLE' '=' ('TRUE' | 'FALSE') NL
```

Note that <use-nagle> is an API version 3.5 option and will lead to an error if it is set in communication with versions < 3.5

8 Installation and Operation

8.1 Installation Instructions for Solaris and Linux

The ESA SLE API software package is delivered as a CD runtime. This CD includes directories containing the API header files, the libraries to link the application program, the API binaries for SLE provider application and for documentation.

For the local SLE API installation the system administrator can decide to copy the supplied files to local installation directories or to use the directories and its contents as the actual installation.

To install the ESA SLE API libraries to a local SLE API installation directory

- Copy all the libraries you need on your local system (see the description of the libraries on section D.5).
- For the shared libraries, a symbolic link shall be made before they can be used, e.g.
- `ln –s libsleapi.so.3 libsleapi.so`
- Update the environment variable `LD_LIBRARY_PATH` if necessary. This variable must contain the directory where the SLE API libraries have been installed.

The binaries located in the bin directory of the delivery must be copied on the local system.

On Solaris platform, these binaries use the ESA SLE API shared libraries. So the `LD_LIBRARY_PATH` environment variable must include the `<sleapi-library-path>`.

8.2 Installation Instructions for MS Windows 2003

The ESA SLE API software package is delivered on a CDROM. When the installation CDROM is inserted in the drive, it automatically starts the installation of the API. Please follow the installation instructions to complete the installation successfully. Is Setup does not start automatically, double-click on the Setup program for Windows2003.

Note that prior to the installation of a new API version an existing old version of the API software shall be uninstalled using 'Add/Remove' programs.

For the local SLE API installation the system administrator can decide to copy the supplied files to any local installation directory or to use the created directories and its contents as the actual installation.

Note that the SLE API libraries are delivered in form of Dynamic Link Libraries (DLL). The SLE application needs to access these libraries during start-up. Therefore the path of the DLLs has to be known to the system. This can e.g. be achieved by extending the 'PATH' environment variable to the path where the SLE API DLLs are residing.

The SLE API DLLs make use of the following system libraries:

`MSVCP71.dll` and `MSVCR71.dll`.

These DLLs are part of the delivery and shall only be used if they are not available on the system where the SLE API is being installed.

8.3 Operating Instructions

The following two binaries are part of the delivery:

- `slecs.exe`: the communications server executable file.
- `sledfl.exe`: the default logger executable file, which can be used optionally (if the SLE application does not implement default logging)

These two binaries are needed for a SLE provider application only.

The communications server must be launched before the default logger process.

To be sure to get all the traces and report messages during start-up, it is recommended to launch the default logger process before the application process.

The communications server process takes one mandatory argument, which is the proxy provider configuration file name:

```
slecs.exe –d <proxy provider config file name>
```

An additional optional argument `–v` offers the printout of configuration problems during start-up:

```
slecs.exe –d <proxy provider config file name> -v
```

The default logger process takes one or two argument; the proxy provider configuration file name, and the trace level (optional). If no trace level is given, the traces are not started, and the default logger receives only report messages. Otherwise, the traces are started with the given trace level (0 = low, 1 = medium, 2 = high, 3 = full).

```
sledfl.exe –d <proxy provider config file name> [-t <trace level>]
```

Example:

```
slecs.exe   -d DB/DBProxyProvider.txt
sledfl.exe  -d DB/DBProxyProvider.txt  -t 2
```

Termination of the communication server and the default logger process:

On Solaris and Linux:

The two processes terminate when they receive the signal SIGTERM:

`kill –15 <pid>`, where <pid> is the process id of the process to be terminated

On MS Windows 2003:

Use the Task Manager utility to terminate the processes as follows:

Select the 'Processes' tab, select `slecs.exe` and `sledfl.exe`, then press the right mouse button and select 'End Process'.

8.4 API Development Environment

This section lists the development environment used to develop the ESA SLE API on various platforms.

On Solaris 8 platform:

Operating System:	Solaris 8
C++ Compiler:	SUNWspro 4.2 C++ compiler
C Compiler:	SUNWspro 4.2 C compiler

On Linux platform:

Operating System:	SUSE Linux Enterprise 12.0 (64 bit)
C++ Compiler:	C++ Compiler gcc version 4.8.3
C Compiler:	C Compiler gcc version 4.8.3

Operating System:	SUSE Linux Enterprise 11.0 (32 and 64 bit)
C++ Compiler:	C++ Compiler gcc version 4.3.2
C Compiler:	C Compiler gcc version 4.3.2

On MS Windows 2003 platform:

Operating System:	MS Windows 2003
C++ Compiler:	MS Visual C++ V7.1
C runtime library:	MSVCR71.dll, multithreaded

APPENDIX A
Notes on the TCP Communications Server Process

A.1 Introduction

This appendix provides supplementary information on the TCP communications server process on a SLE service provider system related to

- use of the Responder Identifier in BIND return PDUs;
- use of simulated time by SLE provider applications;
- listening for incoming connect-requests.

An introduction to the communications server process can be found in section 2.3.

A.2 Use of the Responder Identifier

When receiving a BIND invocation PDU, the communications server extracts the service instance identifier and forwards the encoded BIND PDU to the proxy that has registered the service instance identifier. Decoding and checking of the BIND invocation, including access control and authentication are performed by the proxy. If any of the checks fail, the proxy generates the BIND return and uses the authority identifier of the local application defined in the configuration file.

If the communications server receives a BIND invocation with a service instance identifier that is not registered, it performs the following steps:

- if the initiator is not in the list of peer applications, it generates a BIND return with the diagnostic “access denied”;
- if the initiator is registered, and authentication is required, it performs authentication;
- if authentication fails, it resets the TCP connection;
- if authentication succeeds, or is not required, it generates a BIND return with the diagnostics “no such service instance”.

In the BIND return PDU, the communications server process uses the default authority identifier and password of the local application specified in its configuration file.

For production systems, it is expected that all applications on one host will use the same default authority identifier and password. However, on a simulator use of different identifiers and passwords might be required. In this case, it could happen that the initiator does not expect the responder identifier inserted by the communications server and issues an error. It is stressed that this can only happen when the BIND invocation must be rejected because the service instance identifier is not known.

A.3 Use of Simulated Time by SLE Provider Applications

The API components integrated into SLE applications allow supply of an external time source. The time provided by that time source might be offset from system time by some fixed amount. In contrast, the communications server process always runs on system time.

As indicated in section A.2, the communications server process does not check any of the PDUs received if it can route the initial BIND invocation. Therefore it is not concerned with time stamps in the SLE PDUs. The transport mapping layer implemented by the communications server only uses elapsed timers, such that it is not affected by the time offset.

However, if the communications server process needs to reject a BIND invocation because the service instance identifier is not registered, it might have to authenticate the initiator credentials. In this case it needs to check that the time stamp delivered with the credentials is not older than allowed by the configuration parameter “maximum authentication delay”. If the time offset is larger than that value, authentication might fail.

It is noted that the API does not time tag log records, notifications, and trace records, but expects the time tag to be added by the interfaces supplied by the application. Log records, notifications, and trace records generated by the communications server process are routed to the SLE application process handling the service instance or to the special default logger interface described in section 5.3. It is responsibility of the receiving application to generate consistent time tags.

A.4 Listening for Incoming Connect-Requests

The configuration file must contain all logical port identifiers and the associated TCP sockets on which the communications server process shall accept TCP connections.

The communications server process reads the configuration file on start-up and initialises its internal data structures. However it does not start listening for TCP connection-requests until the first application creates and configures a service instance using a given port identifier. The communications server process stops listening for connection-requests on a port when the last service instance that used a port has been deleted.

APPENDIX B
SLE API Log Messages

The following table gives the description of all the log messages issued by components of the ESA SLE API Package.

Most of the log messages contain placeholders for parameters (e.g <P1>), which are filled at runtime.

For each log message, the type is described (“A” = alarm, “I” = information message).

Message ID	Type	Message Text
1001	A	Protocol Abort This message is only issued if the Down Call Wrapper is used.
1002	A	Operation rejected. <P1> <P2> An operation received from the network has been rejected by the proxy. P1 = diagnostic P2 = operation
1003	A	The link to the application process is broken. <P1> P1 = diagnostic.
1004	A	Encoding error: <P1>. Decoding error: <P1> P1 = error details
1006	A	Read configuration file failed, line <P1>, diagnostic <P2>. P1 = The line number in the configuration file P2 = the diagnostic details
1014	A	Bind aborted before delivery. A Bind Invoke operation was aborted due to a PEER-ABORT invocation or a Protocol-Abort.
1015	A	Peer Abort This message is only issued if the Down Call Wrapper is used.
1501	A	No such file: <P1>. P1 = the file name.
1502	A	Open configuration file failed <P1>. P1 = further error details, if applicable.
1503	A	Parsing error <P1>. P1 = error details.
1504	A	Configuration error <P1>. P1 = error details.
1505	A	Add Proxy rejected, invalid state.
1506	I	Protocol Id <P1> not supported.

Message ID	Type	Message Text
		P1 = the protocol identifier.
1507	I	Duplicate Protocol Id <P1>. P1 = the protocol identifier.
1508	A	No proxy registered for <P1>. P1 = the protocol identifier for which no proxy is registered.
1509	I	Proxy not started for <P1>. P1 = the protocol identifier.
1510	A	No proxy started.
1511	A	Access violation by initiator <P1>. P1 = identifier of the Bind initiator.
1513	A	Protocol Error, PDU <P1>, proxy state <P2>. P1 = the name of the PDU for which the error occurred P2 = the state of the proxy.
1514	A	Protocol Error. Event <P1>, originator <P2>, SI state <P3>. P1 = the state event, P2 = the originator of the event, P3 = the service instance state.
1515	I	Return timer expired.
1516	I	End of service provision period.
1517	I	End of service provision period (Unbind with ‘end’).
1520	I	Transfer buffer discarded.
1522	I	<P1> status report requested. P1 = Start, Periodic, or Stop.
1524	A	Unexpected PDU in concatenation buffer: <P1>. P1 = the name of the PDU.
1526	A	Empty transfer buffer received.
1527	A	Inconsistent or incomplete invocation argument(s): <P1>. P1 = the name of the PDU.
1528	A	Inconsistent or incomplete return argument(s): <P1>. P1 = the name of the PDU.
1529	A	Incompatible invocation PDU received: <P1>. P1 = the name of the PDU.
1530	A	Incompatible return PDU received: <P1>. P1 = the name of the PDU.

Message ID	Type	Message Text
2000	A	Unexpected list element <P1>. An element of a list in the configuration file has been detected, which is not member of the list. P1 = the name of the list element
2001	A	Value for <P1> missing. P1 = the name of a configuration file parameter
2002	A	Value for <P1> already set. P1 = the name of a configuration file parameter
2003	A	Unknown keyword <P1>. P1 = the keyword read from the configuration file
2004	A	Duplicate value <P1> for <P2>. P1 = the value P2 = the parameter for which the duplicate value has been encountered
3000	A	Unable to listen, <P1> is unknown. P1 = the port identifier
3001	A	TCP listen failed for port <P1>, <P2>. P1 = the port identifier P2 = the error details
3002	A	Cannot connect to port <P1>, <P2>. P1 = the port identifier P2 = the error details
3003	A	Error connecting to port <P1>, <P2>. P1 = the port identifier P2 = the error details
3004	A	Cannot write context message, <P1>. P1 = the error details
3006	A	Bad context message.
3007	A	Bad heartbeat parameters.
3008	A	Start-up timeout.
3009	A	Heartbeat receive timeout.
3010	A	Disconnect timeout.
3011	A	Unexpected data after last PDU.
3012	A	Unexpected urgent data.
3013	A	TCP read/write error, <P1>. P1 = the TCP error details during disconnect
3014	A	Unexpected disconnect by peer.

Message ID	Type	Message Text
3015	A	TCP read/write error, <P1>. P1 = the TCP error details
3016	A	Bad header received.
3017	A	Urgent data not received.
3019	A	TCP read error when in abort, <P1>. P1 = the error details
3020	A	TCP write error when in abort, <P1>. P1 = the error details
3021	A	Abort timeout.
3027	A	Local abort ignored in closing state.
3028	A	Local abort ignored in closing state.
3029	A	Send request ignored in closing state.
3030	A	Abort received in closing state, <P1> P1 = the error details
3032	A	TCP accept failed, <P1> P1 = the error details
3034	A	API internal problem (<P1>, <P2>) If this internal error is logged, maintenance shall be called P1 = the name of the PDU P2 = the state of the association
3035	A	TCP read error, <P1> P1 = the error details
3036	A	Unexpected close

APPENDIX C
SLE API Trace Messages

For each trace level, the description of the trace records that are generated by components of the ESA SLE API Package is given.

Service Instance Trace Records	Low	Medium	High	Full
Every state change in the Service Instance	X	X	X	X
Invoke operations received by the Service Instance		X	X	X
Return operations received by the Service Instance		X	X	X
Invoke operations sent to the Proxy		X	X	X
Return operations sent to the Proxy		X	X	X
Invoke operations sent to the Application		X	X	X
Return operations sent to the application		X	X	X
Transfer buffer queuing		X	X	X
Transfer buffer transmission		X	X	X
Transfer Buffer discarding		X	X	X
Reaching the latency limit		X	X	X
End of provision period		X	X	X
Resume Data transfer		X	X	X
Every event in the state machine		X	X	X
Sending of periodic status reports		X	X	X
Full printout of invoke operations			X	X
Full printout of return operations			X	X

Service Protocol Layer Trace Records	Low	Medium	High	Full
Sending of a Connect request to Transport Mapping Layer	X	X	X	X
Reception of a Connect indication		X	X	X
Every state change in the Association		X	X	X
Unable to queue the operation. The queue is full			X	X
Queuing of operations			X	X
Dump of the PDU				X

Communication Server Trace Records	Low	Medium	High	Full
Error of decoding a Bind PDU	X	X	X	X
Bind Operation received with invalid service type	X	X	X	x
Bind Operation received with not supported service type	X	X	X	X
Send a negative Bind Return PDU	X	X	X	X
Reception of a TCP CNX indication		X	X	X

Transport Mapping Layer Trace Records	Low	Medium	High	Full
Start and Stop listening on a port	X	X	X	X
TML connection closed	X	X	X	X
Connection succeeded	X	X	X	X
PDU transmission request		X	X	X
TML requested to send an abort		X	X	X
PDU read			X	X
TML now receiving incoming PDUs			X	X
TML reset the connection			X	X
TML writes the last PDU			X	X
TML receives the last PDU			X	X
TML receives a bad context message			X	X
TML receives a bad header			X	X
Error establishing TML association, timeout occurred			X	X
PDU Transmitted			X	X
Tracing started, and stopped			X	X
Heart Beat message read				X
Heart Beat message transmitted				X
Data Written				X
Timeout occurred				X
PDU header read				X
Context Message received				X
Crossing of abort requests				X

APPENDIX D
Contents of the Distribution

D.1 Overview

The ESA SLE API package is delivered using a set of directories containing different types of files belonging to the API package distribution.

The directories of the distribution are structured as follows:

bin/ : contains the delivered binaries.

doc/ : contains the delivery notes and documents.

include/ : contains all the header files

lib/ : contains a set of SLE API libraries.

LICENSES/ Contains the ESA License text for the SLE API Software, and the Third Party Software License Texts used by the SLE API.

D.2 Contents of the bin/ directory:

The contents of the bin directory depends on the platform:

For the Solaris 8, Linux 32 and Linux 64 delivery:

This directory contains two binaries:

`slecs.exe` The communication server binary

`sledfl.exe` The default logger

For the MS Windows 2003 delivery:

`slecs.exe` The communication server binary

`sledfl.exe` The default logger

`SLEINF.dll` Infrastructure objects needed in all components of the ESA SLE API

`APIUT.dll` SLE API utilities

`APISE.dll` SLE API service element

`APIOP.dll` SLE API operation objects

`APIPX.dll` SLE API proxy

`SLEEXT.dll` ESA SLE API extensions (see section 5).

D.3 Contents of the doc/ directory:

This directory contains the software release note document. This document describes the contents of the delivery, contains the installation instructions and a list of corrected software problems if applicable.

The ESA SLE API Reference Manual is also part of the distribution.

Furthermore this directory contains examples of the API configuration files for the proxy and the service element.

D.4 Contents of the include/ directory:

SLE_Types.h
SLE_APITypes.h
SLE_RESULT.h
SLE_SCM.h
SLE_SCMTypes.h
IMalloc.h
EE.h
ISLE_AssocFactory.h
ISLE_Bind.h
ISLE_Concurrent.h
ISLE_ConfirmedOperation.h
ISLE_Credentials.h
ISLE_EventMonitor.h
ISLE_EventProcessor.h
ISLE_Locator.h
ISLE_Operation.h
ISLE_OperationFactory.h
ISLE_PeerAbort.h
ISLE_ProxyAdmin.h
ISLE_Reporter.h
ISLE_ScheduleStatusReport.h
ISLE_SEAdmin.h
ISLE_SecAttributes.h
ISLE_Sequential.h
ISLE_ServiceInform.h
ISLE_ServiceInitiate.h
ISLE_SIAdmin.h
ISLE_SIFactory.h
ISLE_SII.h
ISLE_SIOpFactory.h
ISLE_SrvProxyInform.h
ISLE_SrvProxyInitiate.h
ISLE_Stop.h
ISLE_Time.h
ISLE_TimeoutProcessor.h
ISLE_TimerHandler.h
ISLE_TimeSource.h
ISLE_Trace.h
ISLE_TraceControl.h
ISLE_TransferBuffer.h
ISLE_Unbind.h
ISLE_UtilFactory.h

CLTU_Types.h
ICLTU_AsyncNotify.h
ICLTU_GetParameter.h
ICLTU_SIAdmin.h
ICLTU_SIUpdate.h
ICLTU_Start.h
ICLTU_StatusReport.h
ICLTU_ThrowEvent.h
ICLTU_TransferData.h

FSP_Types.h
IFSP_AsyncNotify.h
IFSP_TransferData.h
IFSP_FOPMonitor.h
IFSP_GetParameter.h
IFSP_InvokeDirective.h
IFSP_SIAdmin.h
IFSP_SIUpdate.h
IFSP_Start.h
IFSP_StatusReport.h
IFSP_ThrowEvent.h

```
RAF_Types.h
IRAF_GetParameter.h              IRAF_SIAdmin.h
IRAF_SIUpdate.h                  IRAF_TransferData.h
IRAF_Start.h                     IRAF_StatusReport.h
IRAF_SyncNotify.h

RCF_Types.h
IRCF_GetParameter.h              IRCF_SIAdmin.h
IRCF_SITransferBufferControl.h   IRCF_SIUpdate.h
IRCF_Start.h                     IRCF_StatusReport.h
IRCF_SyncNotify.h                IRCF_TransferData.h

ROCF_Types.h
IROCF_GetParameter.h             IROCF_SIAdmin.h
IROCF_SIUpdate.h                 IROCF_Start.h
IROCF_StatusReport.h             IROCF_SyncNotify.h
IROCF_TransferData.h

DCW.h                            DCW_Types.h
IDCW_Admin.h                     IDCW_EventQueue.h
IDCW_SIFactory.h
DFL.h
ESLE_APIBuilder.h                ESLE_Builder.h
ESLE_DefaultLogger.h
ESLE_LoggerCreator.h             ESLE_Reporter.h
ESLE_TimeSource.h
ESLE_Trace.h
```

When the C API is also delivered, the following additional header files are present:

```
EE_BOOL.h                        EE_CAPI_AppClassCreator.h
EE_CAPI_AppClassCreator_c.h      EE_CAPI_CallReporter.h
EE_CAPI_CallTimeSource.h         EE_CAPI_CallTrace.h
EE_MemManage_c.h                 EE_c.h
ESLE_Builder_c.h                 ICLTU_AsyncNotify_c.h
ICLTU_GetParameter_c.h           ICLTU_SIAdmin_c.h
ICLTU_SIUpdate_c.h               ICLTU_Start_c.h
ICLTU_StatusReport_c.h           ICLTU_ThrowEvent_c.h
ICLTU_TransferData_c.h           IDCW_Admin_c.h
IDCW_EventQueue_c.h              IDCW_SIFactory_c.h
```

IRAF_GetParameter_c.h
IRAF_SIAdmin_c.h
IRAF_SIUpdate_c.h
IRAF_Start_c.h
IRAF_StatusReport_c.h
IRAF_SyncNotify_c.h
IRAF_TransferData_c.h
IRCF_GetParameter_c.h
IRCF_SIAdmin_c.h
IRCF_SIUpdate_c.h
IRCF_Start_c.h
IRCF_StatusReport_c.h
IRCF_SyncNotify_c.h
IRCF_TransferData_c.h
ISLE_Bind_c.h
ISLE_ConfirmedOperation_c.h
ISLE_Credentials_c.h
ISLE_OperationFactory_c.h
ISLE_Operation_c.h
ISLE_PeerAbort_c.h
ISLE_Reporter_c.h
ISLE_SEAdmin_c.h
ISLE_SIAdmin_c.h
ISLE_SII_c.h
ISLE_SIOpFactory_c.h
ISLE_ScheduleStatusReport_c.h
ISLE_SecAttributes_c.h
ISLE_ServiceInitiate_c.h
ISLE_Stop_c.h
ISLE_TimeSource_c.h
ISLE_Time_c.h
ISLE_TraceControl_c.h
ISLE_Trace_c.h
ISLE_Unbind_c.h
ISLE_UtilFactory_c.h
IUnknown_c.h

D.5 Contents of the lib/ directory:

The contents of the library directory depends on the platform:

For the Solaris 8 and Linux delivery:

`libsleapi.so.3`	SLE API service element and proxy (dynamic library).
`libsleext.so.3`	ESA SLE API extensions (see section 5) (dynamic library).
`libsleinf.so.3`	Infrastructure objects needed in all components of the ESA SLE API (dynamic library).

When the C API is also delivered, following additional library is present:

`libslec.so.3`	(dynamic library).

For the MS Windows 2003 delivery:

`SLEINF.lib`	Infrastructure objects needed in all components of the ESA SLE API
`APIUT.lib`	SLE API utilities
`APISE.lib`	SLE API service element
`APIOP.lib`	SLE API operation objects
`APIPX.lib`	SLE API proxy
`SLEEXT.lib`	ESA SLE API extensions (see section 5).

When the C API is also delivered, following additional library is present:

`SLEC.lib`	SLE API C interface

)