

IEEE Standard for Information Technology—ACSE and Presentation Layer Services—Application Program Interface (API) [Language Independent]

Sponsor
**Portable Applications Standards Committee
of the
IEEE Computer Society**

Approved September 22, 1994
IEEE Standards Board

Abstract: A language-independent Application Program Interface (API) to the ACSE application service element of the OSI application layer and the connection-oriented services of the presentation layer are defined. The purpose of this API is to support portable implementations of application-specific OSI service elements and non-OSI applications.

Keywords: ACSE, API, application layer, application portability, language independent, open systems, open systems interconnection, OSI, presentation layer

The Institute of Electrical and Electronics Engineers, Inc.

345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1994 by the

Institute of Electrical and Electronic Engineers, Inc.

All rights reserved. Published 1994

Printed in the United States of America.

ISBN 1-55937-465-9

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE standards documents may involve the use of patented technology. Their approval by the Institute of Electrical and Electronics Engineers, Inc. does not mean that using such technology for the purpose of conforming to such standards is authorized by the patent owner. It is the obligation of the user of such technology to obtain all necessary permissions.

Introduction

(This Introduction is not a normative part of IEEE Std 1351-1994, IEEE Standard for Information Technology—ACSE and Presentation Layer Services—Application Program Interface (API) [Language Independent], but is included for information only.)

Purpose

This standard defines a language-independent Application Program Interface (API) to the ACSE application service element from the OSI Application Layer and the connection-oriented services of the OSI Presentation Layer. The purpose of the API is to support portable implementations of application-specific OSI services and non-OSI applications.

The requirements for an implementation of the API and the underlying protocol implementation to which it provides access are defined in clause 1.3. It is not the purpose of this standard to define a particular subset of the ACSE and Presentation Layer protocols which implementations must support.

This standard has two specific groups of implementors as its target audience.

- *API Implementors.* System vendors who are implementing an OSI stack may use this standard to design a portability interface to the services of a protocol stack, facilitating support of applications from diverse sources.
- *Applications Implementors.* Implementors of OSI and non-OSI applications that are to run over OSI protocol stacks may use this standard to assist in the design of applications that are portable across OSI protocol stack implementations from different system vendors. Here the term OSI applications includes both application utilities and applications-specific APIs.

Related Standards

This standard is intended to provide the basis for the definition of programming language bindings to which implementations and applications can conform. Such a language binding, for the C programming language, is contained in IEEE Std 1353-1994 [B4].

Overview

This API provides a primitive-level interface to the connection-oriented services of the Presentation Layer and the ACSE application-service-element, and is designed to support portable implementations of application-specific OSI services and non-OSI application.

The rationale for defining an API to access the services provided by the upper layers of the OSI stack is as follows:

- Above the Transport Layer, the OSI layer and service elements are divided into those that provide services common to all applications (for example, the Presentation Layer, ACSE, etc.), and those that are specific to a particular application or group of applications (for example, FTAM, X.500, etc.). In the absence of an open API to the upper layers of the OSI protocol stack, the application implementor is faced with two choices: either each application must incorporate an implementation of the upper layers; or applications must be implemented to access the upper layers via a nonstandard (perhaps proprietary) API. The former approach does have the advantage of allowing the application to tune the upper layer implementation by closely coupling it to the requirements of an application; however, it has the disadvantage that the user may have to use and manage multiple implementations of the upper layers. The latter approach risks locking the user into a particular product family. A standardized interface at the highest point of commonality shared among most Application Layer services (which is the ACSE/Presentation level) allows the user to avoid this problem by making possible separation between the application-specific and common elements of the stack via an open API.

- Further, an organization that has implemented or purchased an OSI stack may not wish to implement each new OSI Application Layer service as it emerges. A standard interface capable of supporting these applications (like FTAM, X.500, etc.) makes it possible for system vendors and users to purchase Application Layer products from independent software vendors and add them to the existing common stack for lower layers.
 - Finally, there is a requirement to implement non-OSI applications directly over an OSI product stack. This requirement arises either because of the lack of an OSI application service suitable for supporting the particular application, or because the application already incorporates all the Application Layer services it requires. Many non-OSI applications are currently implemented using the services of the OSI Transport Layer. However, a number of characteristics of the transport service (for example, its lack of an orderly release service) make it inappropriate for the support of certain applications without some enhancement. Again, a standardized interface to the upper layers of the OSI stack provides a suitable platform upon which to implement such applications in a portable fashion.
- This latter requirement is an important element in strategies for coexistence between OSI and other protocol suites, and for migration of networks from those protocol suites to OSI, as a number of techniques employed rely upon implementing existing applications on top of an OSI protocol stack.

Figure 1 illustrates possible interfaces to OSI services and their relationship to potential applications.

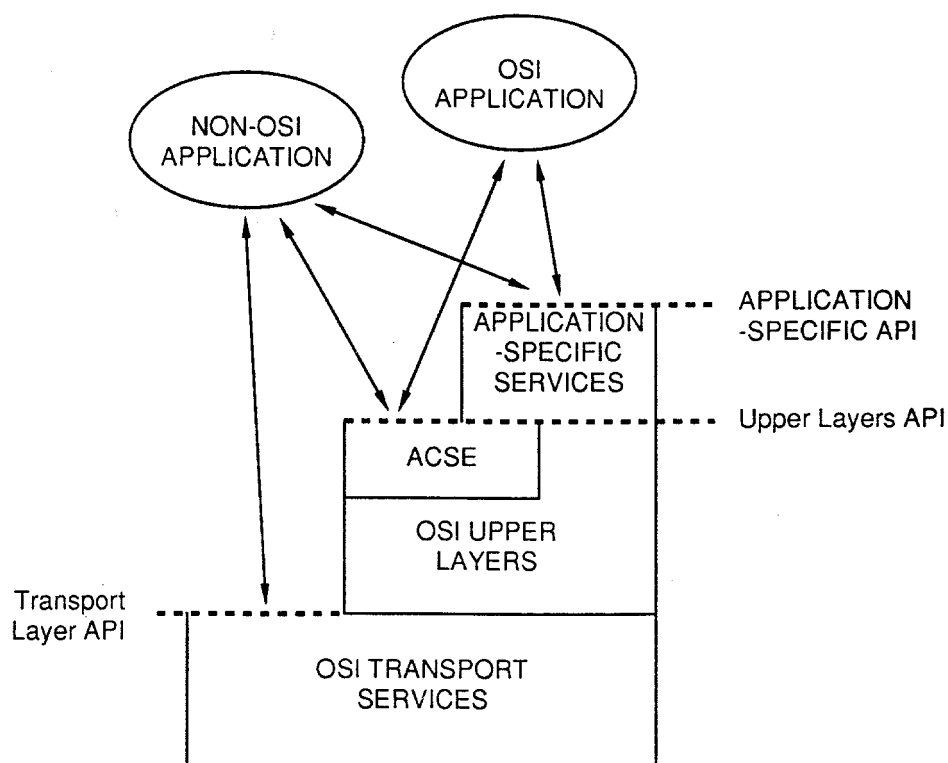


Figure 1 — OSI Service Interfaces

Base Documents

Many of the interfaces and utilities of this standard were adapted from materials in machine-readable forms donated by the following organization:

- The X/Open Company, Ltd., United Kingdom:¹ CAE Specification ACSE/Presentation Services API (XAP). Copyright © September 1993, X/Open Company Limited; reprinted with permission.

Related Standards Activities

There are no related standards activities at this time.²

Extensions are approved as "amendments" or "revisions" to this document, following the IEEE Standards procedures.

Approved supplements are published separately and distributed with orders from the IEEE for this document until the full document is reprinted and such supplements are incorporated in their proper positions.

If you have any questions about the completeness of your version, you may contact the IEEE Standards Department [+1 (908) 562-3800] to determine what supplements have been published.

If you have an interest in participation in the working groups addressing these issues, please send your name, address, and phone number to the Secretary, IEEE Standards Board, Institute of Electrical and Electronics Engineers, Inc., P.O. Box 1331, 445 Hoes Lane, Piscataway, NJ 08855-1331, USA, and ask to have this forwarded to the chairperson of the appropriate PASC working group.

This standard was prepared by the P1238 Working Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. At the time this standard was approved, the membership of the P1238 Working Group was as follows:

Portable Applications Standards Committee (PASC)

Chair: Lowell Johnson
Vice Chairs: Barry Needham
Lorraine Kevra
Andrew Josey
Treasurer: Pete Meier
Secretary: Charles Severance

P1238 Working Group Officials

Chair: Tim Boland
Editor/Secretary: Graham Jack

¹The IEEE is grateful to X/Open for permission to use their machine-readable materials.

²A *Standards Status Report* that lists all current IEEE Computer Society standards projects is available from the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC 20036-1903, USA: Telephone: +1 202 371-0101; Fax: +1 202 728-9614.

Working Group

Reed Adams	Steve Greismer	Curtis Royster
Dave Cannon	Petr Janecek	Larry Spieler
	Ed Kelly	

The following persons were members of the P1238 Balloting Group that approved the standard for submission to the IEEE Standards Board:

Reed Adams	Chris Harding	Jon F. Spencer
Andy Bihain	Jeffrey S. Haemer	Dennis C. Stewart
Tim Boland	Petr Janecek	Mark Sylvester
Loren E. Buhle	Judy Kerner	Mark-Rene Uchida
Malcom Butler	Martin J. Kirk	Angel Vina
Steve Case	Thomas M. Kurihara	Andrew Walker
Philip Enslow, Jr.	Roland McGrath	Keith S. Weir
Steve Griesmer	Curtis Royster	Oren Yuen

When the IEEE Standards Board approved this standard on September 22, 1994, it had the following membership:

Wallace S. Read, *Chair*
Donald C. Loughry, *Vice Chair*
Andrew G. Salem, *Secretary*

Gilles A. Baril	Donald N. Heirman	D. N. "Jim" Logothetis
Bruce B. Barrow	Richard J. Holleman	L. Bruce McClung
Jose A. Berrios de la Paz	Jim Isaak	Marco W. Migliaro
Clyde R. Camp	Ben C. Johnson	Mary Lou Padgett
James Costantino	Sonny Kasturi	Arthur K. Reilly
Stephen L. Diamond	Lorraine C. Kevra	Ronald H. Reimer
Donald C. Fleckenstein	E. G. "Al" Kiener	Gary S. Robinson
Jay Forster*	Ivor N. Knight	Leonard L. Tripp
Ramiro Garcia	Joseph L. Koepfinger*	

* Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal	Richard B. Engelman	David E. Soffrin
James Beall	Robert E. Hebner	
	Rachel A. Meisel	
	<i>IEEE Standards Project Editor</i>	

CLAUSE	PAGE
1. General	1
1.1 Scope	1
1.2 Normative References	2
1.3 Conformance	3
1.4 Test Methods	6
2. Terminology and General Requirements	6
2.1 Conventions	6
2.2 Definitions	8
3. Overview (Informative)	11
3.1 APS API Model	11
3.2 APS Operations	14
3.3 APS Mechanisms	14
3.4 Using the APS API	23
4. APS Environment	24
4.1 API Attribute Descriptions	24
4.2 Environment Attribute Relationships	35
4.3 Environment Attribute Summary	35
5. Service Interface	42
5.1 Introduction	42
5.2 Datatypes	42
5.3 Operations	65
5.4 Client-Supplied Operations	96
5.5 Error Codes	98
6. Primitives	100
6.1 ACSE Primitives	100
6.2 Presentation Primitives—Data Transfer	111
6.3 Presentation Primitives—Dialog Control	113
6.4 State Transitions—Outbound Primitives	125
6.5 Attribute Relationships—Outbound Primitives	128
6.6 State Transitions—Inbound Primitives	130
6.7 Attribute Relationships—Inbound Primitives	133
7. Environment Initialization Interface	136
7.1 Initialization Source File Format	136
7.2 ap_osic — Create an Environment Initialization File	142
7.3 Output Files	143
Annex A Overview of the OSI Upper Layers (Informative)	144
Annex B Bibliography (Informative)	147

IEEE Standard for Information Technology—ACSE and Presentation Layer Services—Application Program Interface (API) [Language Independent]

1. General

1.1 Scope

This standard defines an API providing access to the ACSE Application Service Element of the Application Layer and the Presentation Layer of OSI. The API provides an interface to the service primitives of the OSI association control service element, {2} and {3}; the connection oriented presentation service, {4} and {5}; and the connection oriented session service, {8} and {11}.

In addition, the amendments to the Session Layer standards that address additional synchronization functionality, {10} and {13}; and unlimited user data, {9} and {12}, are within scope.

The services of the connectionless presentation service are outside the scope of this standard.

The standard provides a programming interface, supporting applications that wish to access the services of the OSI ASCE application service element and the connection-oriented presentation service, and further, wish to do so in a way that permits portability among disparate implementations of those services.

An API at the selected layers of the OSI Basic Reference Model is required to support at least two distinct groups of application implementors:

- 1) Independent software vendors who wish to implement OSI application layer products in such a way that they can be run across a wide range of hardware and software platforms, thus increasing the applicability and value of their products.
- 2) Application developers who require direct access to the service primitives at these layers in order to implement connection-oriented applications that require a detailed level of configuration and control of associations.

This standard provides a language-independent specification of an interface and environment to support application portability at the source-code level. It is intended to be used by application developers, system implementors, test method writers, and users.

This standard describes the external characteristics and facilities that are of importance to applications developers, rather than the internal construction techniques employed to achieve these capabilities.

1.2 Normative References

The following standards contain provisions that, through references in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below.

- [1] ISO 7498 : 1984 ¹, *Information processing systems—Open Systems Interconnection—Basic reference model*.
- [2] ISO 8649 : 1988 (E), *Information processing systems—Open Systems Interconnection—Service definition for the association control service element*.
- [3] ISO 8650 : 1988 (E), *Information processing systems—Open Systems Interconnection—Protocol specification for the association control service element*.
- [4] ISO 8822 : 1988 (E), *Information processing systems—Open Systems Interconnection—Connection oriented presentation service definition*.
- [5] ISO 8823 : 1988, *Information processing systems—Open Systems Interconnection—Connection oriented presentation protocol specification*.
- [6] ISO/IEC 8824 : 1990, *Information technology—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*.
- [7] ISO/IEC 8825 : 1990, *Information Technology—Open Systems Interconnection—Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*.
- [8] ISO 8326 : 1987, *Information processing systems—Open systems interconnection—Basic connection oriented session service definition*.
- [9] ISO 8326 /AD 2 : 1988, *Information processing systems—Open systems interconnection—Basic connection oriented session service definition—Addendum 2: Incorporation of unlimited user data*.
- [10] ISO 8326 /AMD 4 : 1992, *Information processing systems—Open systems interconnection—Basic connection oriented session service definition—Amendment 4: Additional synchronization functionality*.
- [11] ISO 8327 : 1987, *Information processing systems—Open systems interconnection—Basic connection oriented session protocol specification*.
- [12] ISO 8327 /AD 2 : 1988, *Information processing systems—Open systems interconnection—Basic connection oriented session protocol specification—Addendum 2: Incorporation of unlimited user data*.
- [13] ISO 8327 /AMD 3 : 1992, *Information processing systems—Open systems interconnection—Basic connection oriented session protocol specification—Amendment 3: Additional synchronization functionality*.

¹ISO/IEC documents can be obtained from the ISO office, 1 Rue de Varembe, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse.

1.3 Conformance

1.3.1 Implementation Conformance

1.3.1.1 Requirements

A conforming implementation for a programming language binding specification for this standard shall meet all of the following criteria:

- 1) The implementation shall support all required behavior defined in this standard.
- 2) The implementation shall support all required interfaces defined in the programming language binding specification. Those interfaces shall support the behavior described in this standard and in the programming language specification.
- 3) The implementation may provide additional functions or facilities not required by this standard or by the programming language binding specification. Each such nonstandard extension shall be identified as such in the system documentation. Nonstandard extensions, when used, may change the behavior of functions or facilities defined by this standard or by the programming language binding specification. The conformance document shall define an environment in which an application can be run with the behavior specified by this standard and the programming language binding specification. In no case shall such an environment require modification of a strictly conforming application.
- 4) An implementation that claims conformance to this standard shall also comply with the conformance clauses of the ISO/IEC protocol specifications that this standard references. These conformance clauses specify:
 - requirements on combinations of functional units
 - by implication, permitted sequences of primitives

It should be noted that the conformance clause of ISO 8327 : 1987 {11} is currently being revised. When this revision is accepted, the revised Conformance clause will apply.

- 5) The choice of underlying profile determines which ACSE, Presentation, and Session Service primitives are supported. The features of the underlying profile are stated in the appropriate Protocol Implementation Conformance Statement (PICS). PICS Proforma for the ISO/IEC Session, Presentation, and ACSE Protocol Specifications are currently under ballot. Conformance completion of these PICS Proforma shall also apply to implementations claiming conformance to this standard, once these PICs have been approved by ISO/IEC.

1.3.1.2 Documentation

A conformance document with the following information shall be available for an implementation claiming conformance to a programming language binding specification for this standard. The conformance document shall be in two parts. The first part shall have the same structure as this standard, with the information presented in the appropriately numbered sections, clauses, and subclauses. The second part shall have the same structure as the programming language binding specification, with the information presented in the appropriately numbered sections, clauses, and subclauses. The conformance document shall not contain information about extended features or capabilities outside the scope of this standard and the programming language binding specification.

The conformance document shall identify the programming language binding specification to which the implementation conforms.

The conformance document shall contain a statement that indicates the full names, numbers, and dates of the language-independent and programming language binding specification standards that apply.

The conformance document shall state which of the optional features defined in this standard and in the programming language binding specification are supported by the implementation.

The conformance document shall describe the behavior of the implementation for all implementation-defined features defined in this standard and in the programming language binding specification. This requirement shall be met by listing these features and providing either a specific reference to the system documentation or providing full syntax and

semantics of these features. The conformance document may specify the behavior of the implementation for those features where this standard or the programming language binding specification states that implementations may vary or where features are identified as undefined or unspecified.

No specifications other than those specified by this standard and the programming language binding specification shall be present in the conformance document.

The phrases “shall document” or “shall be documented” in this standard or in a programming language binding specification for this standard mean that documentation of the feature shall appear in the conformance document, as described previously, unless the system documentation is explicitly mentioned.

The system documentation should also contain the information found in the conformance document.

1.3.1.3 Conforming Implementation Options

The following features of the service are optional:

- 1) Support of the **save_environment** operation is optional. If this operation is invoked and it is not supported by the implementation, the service shall return the error code **feature_not_supported**.
- 2) Support of the **restore_environment** operation is optional. If this operation is invoked and it is not supported by the implementation, the service shall return the error code **feature_not_supported**.
- 3) Support of the **look_at_primitive** operation is optional. If this operation is invoked and it is not supported by the implementation, the service shall return the error code **feature_not_supported**.
- 4) Support of the **control_interrupts** operation is optional. If this operation is invoked and it is not supported by the implementation, the service shall return the error code **feature_not_supported**.
- 5) Support of the **init_environment** operation when called with an *environment_file* argument present is optional. If this feature of the operation is requested and is not supported by the implementation, the service shall return the error code **feature_not_supported**.
- 6) Support of the **ap_osic** command is optional. If this command is invoked and is not supported by the implementation, the result is implementation defined.
NOTE — The result of invoking a command that is not available is usually determined by the operating system environment in which the implementation runs.
- 7) Support of an authorization check to determine whether a client is permitted to use a particular local presentation address is optional. The error code **bind_refused** may be returned by the **bind_address** or **send_primitive** function when a client associated with an APS Instance is not authorized to use the address specified by the **BIND_PADDR** environment attribute.
- 8) If an authorization check is performed to determine if a client is authorized to use the address specified by the **BIND_PADDR** environment attribute, it is an implementation option whether that check is performed when **bind_address** is invoked, or when **send_primitive** is called to send an **A_ASSOC_REQ** or **A_ASSOC_RSP** primitive.
- 9) For both initiating and responding APS Instances, an implementation may support a presentation address that has a zero length nsaps list (known as an NSAP *wildcard address*).
If the **BIND_PADDR** environment attribute contains a presentation address that includes an NSAP wildcard address and this feature is not supported by the implementation, the **bind_address** operation shall return the error code **feature_not_supported**.
- 10) For a responding APS Instance only, an implementation may support a presentation address that has an unspecified S-selector or P-selector—known as a selector *wildcard address*. The implementation shall document which selectors may be left unspecified in a presentation address.

NOTE — It is a requirement of the underlying service that a value for the T-Selector shall be specified.

If the **BIND_PADDR** environment attribute contains a presentation address that includes a selector wildcard address and this feature is not supported by the implementation, the **bind_address** operation shall return the error code **feature_not_supported**.

The following features of the service are implementation defined:

- 1) An implementation shall document which international standardized profiles (ISPs) are supported by the underlying ACSE/Presentation service provider.

NOTE — Currently, there are no ratified ISPs that deal specifically with profiles of the services of the OSI upper layers. Instead, individual application profiles define the options that they require of the ACSE, Presentation, and Session services. An ISP for the OSI upper layers is under development as ISO/IEC pDISP 11188: 1992, Common Upper Layer Requirements {B1}. When this work is ratified as an ISP it will become a suitable means of specifying the features of the underlying service provider.

If the `send_primitive` operation is invoked with the *primitive to send* argument set to a value that identifies a primitive that is not supported by the underlying ACSE/Presentation service provider, the service shall return the error code `invalid_primitive`.

NOTE — This error code may also be returned when a primitive is supported by the underlying ASCE/Presentation service provider, but it is not available for use on the current association due to functional unit selection.

- 2) An implementation shall document which specific transport profiles are supported by the underlying ACSE/Presentation service provider.

NOTE — The ISO application profiles list the range of transport profiles with which they may be combined.

- 3) When multiple network address components are included in a presentation address, the specific network address(es) chosen by the provider and the manner by which it is selected for initiating or listening to connections is implementation defined.

1.3.2 Application Conformance

All applications claiming conformance to a programming language binding specification for this standard shall fall within one of the following categories.

1.3.2.1 Strictly Conforming Application

A Strictly Conforming Application is an application that requires only the facilities described in this standard, in the programming language binding specification, and in the applicable language standards. Such an application shall accept any behavior described in this standard or in the programming language binding specification as unspecified or implementation defined, and for symbolic constants shall accept any value in the ranges permitted by this standard and the programming language binding specification.

1.3.2.2 ISO/IEC Conforming Application

An ISO/IEC Conforming Application of a programming language binding specification for this standard is an application that uses only the facilities described in this standard, in the programming language binding specification, in the applicable language standards, and in other language binding standards approved by ISO or IEC. Such an application shall include a statement of conformance that documents all options and limit dependencies, and all other ISO or IEC standards used.

1.3.2.3 Conforming Application Using Extensions

A Conforming Application Using Extensions of a programming language binding specification for this standard is an application that differs from a Conforming Application only in that it uses nonstandard facilities that are consistent with this standard and with the programming language binding specification. Such an application shall fully document its requirements for these extended facilities, in addition to the documentation required of a Conforming Application.

1.4 Test Methods

There are no test assertions for this standard at present.

2. Terminology and General Requirements

2.1 Conventions

2.1.1 General and Typographical Conventions

Language-independent datatype, operation, argument, and error names are in lowercase and are spelled with underscores between words (for example, `send_primitive`).

Service primitives, environment variables, and APS Instance states are referred to using the names defined by the state datatypes that the service provides to identify them (for example, `A_ASSOC_REQ`, `ROLE_ALLOWED`, and `UNBOUND` respectively).

The use of fonts in this standard is as follows.

- The **Helvetica** font is used for:
 - Language-independent operation names, such as `open_instance`
 - Language-independent datatype names, such as `aeq_type`
 - Language-independent error names, such as `invalid_aps_instance`
- The *italic* font is used for:
 - Language-independent operation arguments, such as *aps_instance*
 - Introducing important terms
 - Crossreferences in 2.2

2.1.2 Language-Independent Conventions

This subclause defines the notation used within this standard to specify the information that is communicated through interfaces to the defined services. The notation assumes a model of an abstract procedural interface, although language bindings to this standard may take other forms.

NOTE — In the absence of a normative notation for language-independent specifications, these conventions are based upon a draft of the POSIX 1003.1 LIS conventions 3. The octet and character types have been added and a syntax has been defined to allow reference to elements of constructed types.

The key elements of the notation are abstract data *datatype* definitions, *name* definitions, and *operation* definitions. The semantics of each of these elements is defined in 2.2.2.

NOTE — The notations specified are supplemented by additional descriptions using other notational schemes (including natural language). For instance, the notation described here is used in the **Synopsis** subclause, while the semantics are described in the **Description** subclause using natural language.

2.1.2.1 Notation

A modified BNF notation is used in this subclause. The characters “:”, “=”, “<”, “>”, “|”, “{”, and “}” are always part of this notation and are never used in terminal strings.

Nonterminal elements consist of a “<” character followed by an identifier followed by a “>” character. Each production consists of a nonterminal followed by the characters “::=” followed by a right-hand side expression. In a

right-hand side expression, the “|” character separates alternate elements, and the “{” and “}” characters delimit optional elements.

2.1.2.2 Identifiers

```

<identifier> ::= <initial-char> <identifier-body>
<initial-char> ::= <uppercase-letter> | <lowercase-letter>
<identifier-body> ::= <body-char> { <identifier-body> }
<body-char> ::= <uppercase-letter> | <lowercase-letter> |
               <digit> | <separator>
<lowercase-letter> ::= a | b | c | d | e | f | g | h | i |
                       j | k | l | m | n | o | p | q | r | s | t |
                       u | v | w | x | y | z
<uppercase-letter> ::= A | B | C | D | E | F | G | H | I |
                       J | K | L | M | N | O | P | Q | R | S | T |
                       U | V | W | X | Y | Z
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
<separator> ::= - | -

```

2.1.2.3 Numbers

```

<number> ::= <digit> | <digit><number>
<signed-number> ::= { - } <number>

```

2.1.2.4 Datatypes

The following notation is used to specify a datatype:

```

Datatype:      <datatype-name>
Definition:    <datatype>
Description:  ...

```

where

```

<datatype> ::= <datatype-name> | <datatype-definition>
<datatype-name> ::= <identifier>
<datatype-definition> ::= <atomic-type> | <constructed-type>
<atomic-type> ::= boolean | <opaque-type>
               <state-type> | <ordinal-type> | <integer-type> |
               <character-type> | <octet-type>
<opaque-type> ::= opaque [ <size> ]
<state-type> ::= state [ <state-list> ]
<state-list> ::= <state-value> { < , state-list> }
<state-value> ::= <identifier>
<ordinal-type> ::= ordinal [ <size> ]
<size> ::= <number>
<integer-type> ::= integer [ <minimum> , <maximum> ]
<minimum> ::= <signed-number>
<maximum> ::= <signed-number>
<octet-type> ::= octet
<character-type> ::= character [ <character-set> ]
<character-set> ::= <identifier>

<constructed-type> ::= <array-type> | <record-type> |
                     <set-type> | <choice-type> | <set-type>

```

```

⟨array-type⟩ ::= array { ⟨index-type⟩ } of ⟨base-type⟩
⟨index-type⟩ ::= ⟨ordinal-type⟩ | ⟨integer-type⟩ | ⟨state-type⟩
⟨base-type⟩ ::= ⟨datatype⟩
⟨record-type⟩ ::= record [ ⟨member-list⟩ ]
⟨choice-type⟩ ::= choice [ ⟨member-list⟩ ]
⟨set-type⟩ ::= set [ ⟨datatype⟩ ]
⟨member-list⟩ ::= ⟨member-definition⟩ { , ⟨member-list⟩ }
⟨member-definition⟩ ::= ⟨member-name⟩ : ⟨member-type⟩
⟨member-name⟩ ::= ⟨identifier⟩
⟨member-type⟩ ::= ⟨datatype⟩

```

Any of the values within square brackets may be listed as *⟨implementation defined⟩*; see 2.2.1.2 for the requirements imposed by this notation.

2.1.2.5 Names

The following notation is used to specify a name:

```

Name:           ⟨identifier⟩
Datatype:         ⟨datatype-definition⟩
Description:      ...

```

2.1.2.6 Operations

The following notation is used to specify a operation:

```

Operation:  ⟨identifier⟩
Status:       ⟨datatype⟩
Input:        ⟨parameter-name⟩ : ⟨datatype⟩
              ⟨parameter-name⟩ : ⟨datatype⟩
              ...
Output:       ⟨parameter-name⟩ : ⟨datatype⟩
              ⟨parameter-name⟩ : ⟨datatype⟩
              ...
Description:  ...

```

where

```

⟨parameter-name⟩ ::= ⟨identifier⟩

```

2.1.2.7 Member Reference

The following notation is used to refer to a member of a record or choice datatype:

```

⟨reference⟩ ::= ⟨base-ref⟩ ⟨element-ref⟩
⟨base-ref⟩ ::= ⟨datatype-name⟩ | ⟨parameter-name⟩
⟨element-ref⟩ ::= ⟨member-name⟩ | ⟨member-name⟩⟨element-ref⟩

```

2.2 Definitions

2.2.1 Terminology

For the purposes of this standard, the following definitions apply:

2.2.1.1 conformance document: A document provided by an implementor that contains implementation details as described in 1.3.1.2

2.2.1.2 implementation defined: An indication that the implementation shall define and document the requirements for correct program constructs and correct data of a value or behavior [ISO/IEC 9945-1 : 1990 {B2}]

2.2.1.3 may: An indication of an optional feature [ISO/IEC 9945-1 : 1990 {B2}].

With respect to implementations, the word *may* is to be interpreted as an optional feature that is not required in this standard, but can be provided. With respect to Strictly Conforming Applications, the word *may* means that the optional feature shall not be used.

2.2.1.4 shall: An indication of a requirement on the implementation or on Strictly Conforming Applications, where appropriate [ISO/IEC 9945-1 : 1990 {B2}].

2.2.1.5 should:

- 1) With respect to implementations, an indication of an implementation recommendation, but not a requirement [ISO/IEC 9945-1 : 1990 {B2}].
- 2) With respect to applications, an indication of a recommended programming practice for applications and a requirement for Strictly Conforming Applications [ISO/IEC 9945-1 : 1990 {B2}].

2.2.1.6 supported: A condition regarding optional functionality.

Certain functionality in this standard is optional, but the interfaces to that functionality are always required. If the functionality is *supported*, the interfaces work as specified by this standard (except that they do not return the error condition indicated for the unsupported case). If the functionality is not *supported*, the interface shall always return the indication specified for this situation [ISO/IEC 9945-1 : 1990 {B2}].

2.2.1.7 system documentation: All documentation provided with an implementation, except the conformance document [ISO/IEC 9945-1 : 1990 {B2}].

Electronically distributed documents for an implementation are considered part of the system documentation.

2.2.1.8 undefined: An indication that this standard imposes no portability requirements on the use of an indeterminate value by an application, or its behavior with erroneous program constructs or erroneous data [ISO/IEC 9945-1 : 1990 {B2}].

Implementations (or other standards) may specify the result of using that value or causing that behavior. An application using such behaviors is using extensions, as defined in 1.3.2.3

2.2.1.9 unspecified: An indication that this standard imposes no portability requirements on applications for correct program constructs or correct data regarding a value or behavior [ISO/IEC 9945-1 : 1990 {B2}].

Implementations (or other standards) may specify the result of using that value or causing that behavior. An application requiring a specific behavior, rather than tolerating any behavior when using that functionality, is using extensions, as defined in 1.3.2.3

2.2.2 General Terms

For the purposes of this standard, the following definitions apply:

2.2.2.1 array: The language-independent syntax for a family of datatypes constructed from a base datatype and an index datatype. The base datatype may be any datatype, the index datatype shall be a finite ordered datatype [TCOS-LIS/D4 {B3}].

2.2.2.2 application-association: A cooperative relationship between two applications for the purpose of communication of information and coordination of their joint operations.

2.2.2.3 association: A synonym for *application-association*.

2.2.2.4 ACSE/Presentation Service (APS) Environment: The collection of information, associated with a particular APS instance, necessary to initiate and maintain an association with another application entity.

2.2.2.5 APS Instance: An instantiation of an APS API service provider, including the APS environment and internal state information.

2.2.2.6 boolean: The language-independent syntax for a datatype with two values, named “TRUE” and “FALSE”, and a set of logical operations: NOT, OR, AND, and so on [TCOS-LIS/D4 {B3}].

2.2.2.7 attribute: A single piece of information stored in the APS environment.

2.2.2.8 choice: The language-independent syntax for a family of datatypes constructed from a sequence of base datatypes, each associated with a name. A value of choice datatype contains, for exactly one name, a value of the corresponding datatype [TCOS-LIS/D4 {B3}].

2.2.2.9 client: Software that uses the interface.

2.2.2.10 datatype: A collection of distinguished values, together with a collection of characterizing operations on those values [TCOS-LIS/D4 {B3}].

2.2.2.11 list: The language-independent syntax for a family of datatypes constructed from a base datatype. A value of list datatype contains a sequence of zero or more values of the base datatype. Applying an operation to all members of a list may be supported through either of two programming paradigms. In the first, the sequencing control is provided by the application. In the second, it is provided by the implementation [TCOS-LIS/D4 {B3}].

2.2.2.12 name: The language-independent syntax for an identifier used to represent a specific value of a specific datatype. When a name is bound to its value is not specified: at compile time, link time, or load time. The use of a name as a parameter in a datatype specification does not imply that its value is defined at compile time [TCOS-LIS/D4 {B3}].

2.2.2.13 opaque: The language-independent syntax for a family of datatypes with no order or other operations defined. An opaque datatype may have associated names that identify distinguished values [TCOS-LIS/D4 {B3}].

2.2.2.14 operation: The language-independent syntax for a program abstraction with formal input and output parameters that are bound to objects or values in the application that invokes it [TCOS-LIS/D4 {B3}].

2.2.2.15 record: The language-independent syntax for a family of datatypes constructed from a sequence of base datatypes, each associated with a name. A value of record datatype contains, for each name, a value of the corresponding base datatypes [TCOS-LIS/D4 {B3}].

2.2.2.16 service: Software that implements the interface.

2.2.2.17 service provider: An implementation of the ACSE and Presentation Layer protocols, to which the APS API provides access.

2.2.2.18 set: The language-independent syntax for a family of datatypes constructed from a base datatype. A value of a set datatype contains an unordered collection of values of the base datatype, each occurring at most once [TCOS-LIS/D4 {B3}].

2.2.2.19 state: The language-independent syntax for a family of unordered, finite datatypes. Each state value is identified by an associated name [TCOS-LIS/D4 {B3}].

2.2.2.20 status datatype: The language-independent syntax for an abstract datatype whose values may be bound to “control” values as well as “data” values [TCOS-LIS/D4 {B3}].

2.2.3 Abbreviations

For the purposes of this standard, the following abbreviations apply:

2.2.3.1 ACSE: Association Control Service Element.

2.2.3.2 AE: Application Entity.

2.2.3.3 AP: Application Process.

2.2.3.4 API: Application Program Interface.

2.2.3.5 APS: ACSE/Presentation Service.

2.2.3.6 APS API: The API defined by this standard..

2.2.3.7 ASN.1: Abstract Syntax Notation One (ISO/IEC 8824:1990 {6}).

2.2.3.8 DCS: Defined Context Set.

2.2.3.9 DPCN: Default Presentation Context Name.

2.2.3.10 DPCR: Default Presentation Context Result.

2.2.3.11 FTAM: File Transfer, Access, and Management.

2.2.3.12 NSAP: Network Service Access Point.

2.2.3.13 OSI: Open Systems Interconnection.

2.2.3.14 PCDL: Presentation Context Definition List.

2.2.3.15 PCDRL: Presentation Context Definition Result List.

2.2.3.16 PDU: Protocol Data Unit.

3. Overview (Informative)

NOTE — This section provides an overview of the APS API and is not Normative. It does not contain requirements for conforming implementations. It describes the model upon which the API is based, its operations, and the mechanisms provided for communication with it. A brief example of how the interface may be used to establish an association and transfer data is also provided.

3.1 APS API Model

Figure 3-1 shows the main features of the model upon which the ACSE Presentation Service API is based.

The API provides operations for establishing associations and transferring data using ACSE and Presentation Layer service primitives, and an *environment* that is used to store information for use in the current association. Requests and responses from the *client* are combined with information from the environment and passed as ACSE or Presentation Service primitives to the *service provider*. Indications and confirmations from the service provider update the AP environment and are passed to the client.

The features and terminology of the APS Model are discussed below.

3.1.1 Service Client

The *client* is an application that uses the services of the OSI ACSE and Presentation Layer services to implement its operations. This client corresponds to the concept of an *application entity*, defined by the OSI reference model, ISO 7498 : 1984 {1}. Depending on the architecture of the operating system and the requirements of the application, the client may be a single-threaded or multi-threaded program or a group of cooperating processes.

Possible clients are OSI and non-OSI applications (for example, an *FTAM initiator* utility or an application ported from another protocol suite), or an implementation of a higher-level API.

3.1.2 Service Provider

The APS API is a standardized interface to the services of an implementation of the ACSE and Presentation layers, known as the *service provider*. An end-system may support one or more service providers via the API. A particular

service provider is selected by specifying a *service provider identifier* when an APS Instance is created. This specification does not assign any syntax to this identifier, other than that it shall be a character string. Individual implementations may specify additional syntax and semantics. For example, an implementation might require that the identifier is a valid POSIX.1 {B2} pathname, identifying a special file associated with the selected service provider.

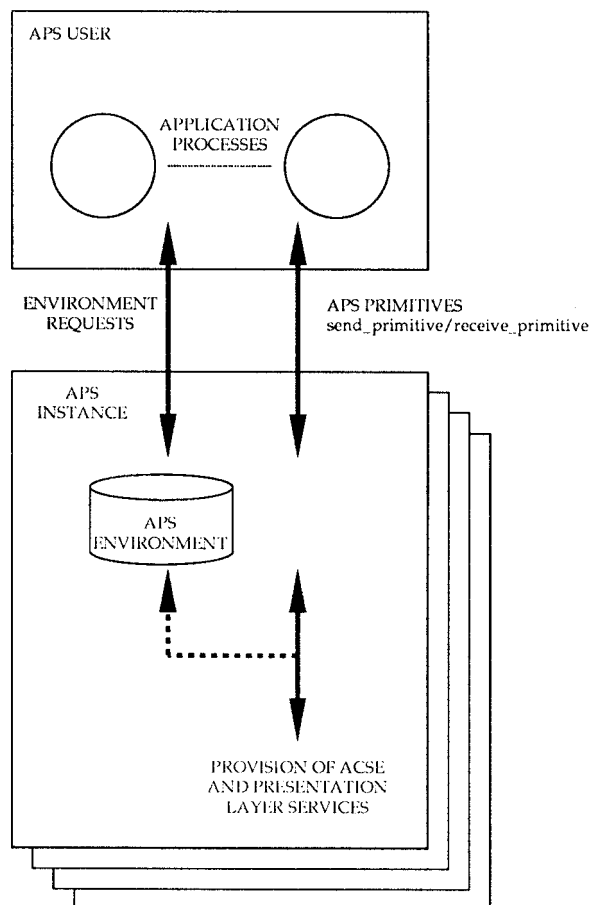


Figure 3-1 – APS Model

This standard does not define explicitly the service options that shall be supported by the provider. The requirements placed upon the service are defined in 1.3.1.3.

3.1.3 APS Instance

An APS Instance is an instantiation of an APS API service provider, including the APS environment and internal state information.

An instance includes the AP *environment* and any internal state information required by the service. A separate APS Instance is created to support each association that a client wishes to establish. The client may have multiple associations established concurrently.

When the instance is created, an *instance identifier* is returned that is used to identify this instance in subsequent invocations of the API operations. (The associated environment shall first be initialized before the instance can be used to send and receive primitives.) This identifier is meaningful only within the context of the process that created the instance. This standard does not define how an instance is passed from one process to another.

3.1.4 APS Environment

The APS environment is a collection of information, associated with a particular APS instance, necessary to initiate and maintain an association with another application entity.

A single piece of information stored in the environment is referred to as an *attribute*. A client can read or modify characteristics of an association or the OSI protocol stack that supports it by performing operations on these attributes.

Many of these attributes correspond to parameters associated with the ACSE service primitives. Thus, the environment provides a mechanism for setting information that is to be used when establishing an association. See 3.1.5 for more details.

There are two types of attributes: read-write attributes and read-only attributes. Read-only attributes cannot, be set by the client. They typically store either static information (for example, available protocol versions) or status information (for example, the current state of the APS Instance). All other attributes are both readable and writable by the application.

It should be noted that the classification of an attribute as *read-only* or *read-write* does not imply that those operations may be performed on it at any time. To the contrary, some *read-only* attributes are readable only in certain states. Similarly, *read-write* attributes are not necessarily readable and writable in every or even the same states. Rather they are distinguished from *read-only* attributes by the fact that they are writable in some state. Complete information about the readability and writability of each attribute is provided in section 4.

3.1.5 Service Primitive Parameters

With the exception of the *user data* parameter, the parameters passed to the service provider when sending a particular primitive are derived from attributes held in the environment and from *primitive parameter data* passed to the service by the client with each primitive request. The parameters passed with an invocation of **send_primitive** allows the client to set parameters specific to each primitive and to override some of the values held in the environment when establishing an association.

When receiving primitives from the service provider, the service updates the values of some environment attributes and passes specified parameters to the client along with the notification of the primitive itself.

For example, when an A_ASSOC_IND primitive is received, the values of several attributes may change. One such attribute is PCDDL, which is set to reflect the presentation context definition list specified by the application requesting the association. The relationship of each primitive to the environment is discussed for each primitive in section 6.

3.1.6 User Data

Almost all primitives to which the service provides access accept a *user data* parameter. For OSI protocols, the content of *user data* is generally a protocol data unit (PDU) for a higher layer protocol (for example, the FTAM protocol includes an F-INITIALIZE PDU in an A-ASSOCIATE request by passing it to ACSE as the *user information* parameter). For non-OSI protocols, the content of *user data* is not constrained.

Unlike other parameters, *user data* is encoded and decoded by the client, according to the requirements of the ACSE and Presentation Layer protocols. General requirements for the encoding and decoding of user data are stated by the descriptions of the **receive_primitive** and **send_primitive** operations. The descriptions of some primitives, given in section 6, indicate specific requirements for encoding of user data.

3.2 APS Operations

The APS API defines the following operations:

- Operations used to establish and release an APS Instance: `open_instance`, `close_instance`.
- Operations used to manage the AP environment: `init_environment`, `set_attribute_value`, `get_attribute_value`.
- Operations used to bind a presentation address with an APS Instance: `bind_address`.
- Operations used to send/receive primitives: `send_primitive`, `receive_primitive`, `look_at_primitive`.
- Operations used to support asynchronous mode operations: `control_interrupts`, `poll_events`.
- Operations used to facilitate sharing an APS Instance: `save_environment`, `restore_environment`.
- Operations to generate error messages from error codes: `error_text`.

3.3 APS Mechanisms

3.3.1 Establishing and Releasing an APS Instance

In order to use the APS API, the client first creates an APS Instance. This is accomplished by using the `open_instance` operation. It is possible that an instance may also be obtained through some other mechanism. However, such mechanisms depend upon the details of a specific implementation and are thus outside the scope of this standard. As part of the `open_instance` operation, the client identifies the particular service provider that is to support this APS Instance. This standard defines the provider identifier as an uninterpreted string. Individual implementations may assign additional semantics to this string according to local conventions.

NOTE — For example, an implementation that runs under a POSIX-conformant operating system may require this string to be a *pathname* identifying a *special file* that is associated with the service provider.

The operation returns an integer that is used to identify this instance in subsequent interactions with the service.

The `close_instance` operation is used to indicate that an instance is no longer needed and that the resources required to support it can be returned to the system.

3.3.2 Reuse of an APS Instance

An APS Instance may be used to support a sequence of application-associations (but only a single association may exist at any one time). After an application-association is terminated (normally or abnormally) the state of the APS Instance is set to IDLE. The instance can then be closed or used to establish another association. Prior to establishing another association, the client may reset those environment attributes that are writable in the IDLE state.

NOTE — Changes to the `BIND_PADDR` variable only take effect when the `bind_address` operation is invoked.

3.3.3 Managing the Environment

Three operations are provided to perform operations on the AP environment.

- `init_environment` allocates space for the environment, if required, and initializes the attributes.
- `set_attribute_value` sets a particular (writable) attribute to a specified value.
- `get_attribute_value` retrieves the value of a particular (readable) attribute.

3.3.4 Sending and Receiving Primitives

The services offered by the ACSE Presentation Layer service provider are made available to the client through a collection of primitives that are sent and received using the `send_primitive` and `receive_primitive` operations. The ACSE and Presentation Layer services available are shown in Table 3-1 together with the related service primitives.

Table 3-1—Primitives Supported by the Service

Services	Service Primitives			
	Send		Receive	
A-ASSOCIATE	A_ASSOC_REQ	A_ASSOC_RSP	A_ASSOC_IND	A_ASSOC_CNF
A-RELEASE	A_RELEASE_REQ	A_RELEASE_RSP	A_RELEASE_IND	A_RELEASE_CNF
A-ABORT	A_ABORT_REQ		A_ABORT_IND	
A-P-ABORT	A_P_ABORT_REQ		A_P_ABORT_IND	
P-DATA	P_DATA_REQ		P_DATA_IND	
P-CAPABILITY-DATA	P_C_DATA_REQ	P_C_DATA_RSP	P_C_DATA_IND	P_C_DATA_CNF
P-TYPED-DATA	P_T_DATA_REQ		P_T_DATA_IND	
P-EXPEDITED-DATA	P_X_DATA_REQ		P_X_DATA_IND	
P-TOKEN-GIVE	P_TOKENGIVE_REQ		P_TOKENGIVE_IND	
P-TOKEN-PLEASE	P_TOKENPLEASE_REQ		P_TOKEN_PLEASE_IND	
P-CONTROL-GIVE	P_CTRLGIVE_REQ		P_CTRLGIVE_IND	
P-SYNC-MINOR	P_SYNCMINOR_REQ	P_SYNCMINOR_RSP	P_SYNCMINOR_IND	D_SYNCMINOR_CNF
P-SYNC-MAJOR	P_SYNCMAJOR_REQ	P_SYNCMAJOR_RSP	P_SYNCMAJOR_IND	P_SYNCMAJOR_CNF
P-RESYNCHRONIZE	P_RESYNC_REQ	P_RESYNC_RSP	P_RESYNC_IND	P_RESYNC_CNF
P-U-EXCEPTION-REPORT	P_UXREPORT_REQ		P_UXREPORT_IND	
P-P-EXCEPTION-REPORT			P_PXREPORT_IND	
P-ACTIVITY-START	P_ACSTART_REQ		P_ACSTART_IND	
P-ACTIVITY-RESUME	P_ACTRESUME_REQ		P_ACTRESUME_IND	
P-ACTIVITY-END	P_ACTEND_REQ	P_ACTEND_RSP	P_ACTEND_IND	P_ACTEND_CNF
P-ACTIVITY-INTERRUPT	P_ACTINTR_REQ	P_ACTINTR_RSP	P_ACTINTR_IND	P_ACTINTR_CNF
P-ACTIVITY-DISCARD	P_ACTDISCARD_REQ	P_ACTDISCARD_RSP	P_ACTDISCARD_IND	P_ACTDISCARD_CNF

NOTE — While there is no P-P-ABORT request service defined in ISO 8822 : 1988 {4}, the A_PABORT_REQ primitive is provided to allow the service user to make it appear as though the Presentation Layer aborted the association in the case where a decoding error is detected. (As discussed later, the client is responsible for encoding and decoding certain data values.)

Services that can be initiated by the client may be associated with either two or four service primitives, depending on whether or not the service is confirmed. Figure 3-2 illustrates how an initiator and a responder may use the `send_primitive` and `receive_primitive` operations, together with the A_ASSOC primitives, to establish an association.

Information about the effects on an APS Instance of sending and receiving each of the primitives is provided in section 6.

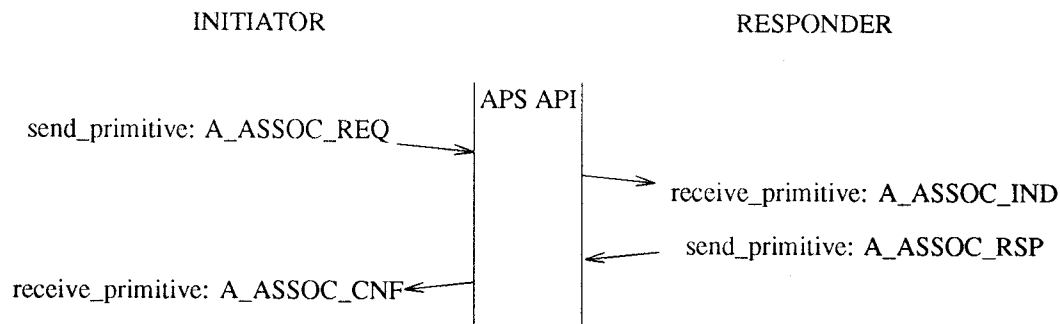


Figure 3-2—Example Primitive Exchange

3.3.5 Sharing an APS Instance

In some implementations, the memory allocated for storing APS environment and state information is located in the data space of the process that created the instance. Should this process wish to pass the instance to another process, data from the address space of the first process has to be copied to the address space of the second.

To facilitate sharing an APS Instance among cooperating processes, `save_environment` and `restore_environment` operations are provided.

A common use of these operations is to support passing of an APS Instance from a process to another, such as from a process to one of its children. Another possible use is to save default settings for a group of applications or a particular network environment. The resulting file can be used to initialize the environment of an AP Instance prior to setting attributes specific to the association being created.

The `save_environment` operation writes a “snapshot” of the storage associated with an APS Instance to a file that was opened by the client. The file permission and file and record locking capabilities of the operating system can be used to control access to this file.

The `restore_environment` operation reloads the snapshot into an APS Instance. This recreates for the referenced instance, the environment and state as it was when the snapshot was saved by the call to `save_environment`. The APS Instance into which the snapshot is restored may have been created directly by the calling user or may have been transferred from another process (the mechanism by which this transfer is achieved is not specified in this standard). Once the restore has been performed the client can resume operations using the APS Instance from the point where the “saving” process left off.

If `restore_environment` is called for a transferred instance, the instance shall be in the same state as it was when the snapshot was saved. No events may be sent or received in the intervening period. It should be noted that this means that `save_environment` and `restore_environment` cannot be used to “roll back” the state of the ACSE and Presentation protocol machines. Events that were processed after an environment was saved cannot be replayed by restoring the APS Instance to its former state.

On the other hand, if the operations are used to initialize an APS Instance, the instance shall be in either the IDLE or UNBOUND state both when the snapshot is saved and when it is restored.

3.3.6 Association Listeners

An Association Listener is an application that uses an APS Instance to examine incoming `A_ASSOC_IND` primitives and to pass the APS instance to a suitable application that is to handle the incoming association.

Support for association listeners is optional. Depending on the application requirements and the optional APS operations available, an association listener may be implemented in various ways. An implementation that is to support association listening is encouraged to support it in one of the following ways.

- *A simple listener mechanism using no optional APS functions:* A listener application creates an APS instance using the **open_instance** operation, binds to an address and waits for an incoming A_ASSOC_IND. It receives the association request using the **receive_primitive** operation, examines the primitive parameters to determine the appropriate receiving application to handle the association, and passes the instance to it in an implementation-specific mechanism. The listener can then close the APS instance by calling **close_instance** to release local resources.

NOTE — The mechanism by which the responding application receives the *aps_instance* identifier, the APS environment of the listener, and the parameters and data associated with the received A_ASSOC_IND primitive is implementation-defined.

- *A listener mechanism making use of the optional APS operations:* A listening application creates an APS instance using **open_instance**, binds to an address and waits for an incoming A_ASSOC_IND. It invokes **look_at_primitive** to obtain the parameters and possibly user data associated with the A_ASSOC_IND primitive, examines the primitive parameters to determine the appropriate receiving application to handle the association, and passes the instance to it in an implementation-specific mechanism. The listener can then close the APS instance by calling **close_instance** to release local resources.

The receiving application calls *ap_restore()*, with a NULL file argument to: create an APS environment; retrieve the local presentation address to which the A_ASSOC_IND primitive was addressed; and set the APS instance state to IDLE.

The receiving application is now in a state where it can invoke the **receive_primitive** operation to receive the pending A_ASSOC_IND primitive.

NOTE — The mechanism by which the responding application receives the *aps_instance* identifier is implementation-defined.

3.3.7 Presentation Context Negotiation

An important aspect of association establishment is negotiation of the presentation contexts to be used during the association. When an association is to be initiated, the client shall set the PCDL and DPCN attributes to propose any presentation contexts that are to be used for exchanging user data (the service provider ensures that the presentation context used for exchange of ACSE protocol information between peer entities is available; however, the user may include it if required).

Similarly, in responding to an association request, the client shall indicate which proposed presentation contexts are acceptable by setting the PCDRL and DPCR attribute before issuing an A_ASSOC_RSP primitive.

NOTE — All encoding and decoding of user data is performed by the client, the service does not accept or reject any requested context. It is up to the client to do so.

3.3.8 Presentation Addresses

As specified in the OSI Basic Reference Model (ISO 7498 : 1984 {1}), an Application Entity shall be addressable via a single, globally unique presentation address. Thus, each client has a unique presentation address. Multiple APS Instances created by the same client may share the same presentation address.

A client defines its presentation address by setting the BIND_PADDR attribute and then invoking the **bind_address** operation. This attribute shall be set before any service primitives can be sent or received by the client. If the client is the association-initiator, it shall also specify the presentation address of the association-responder (by setting the REM_PADDR attribute) prior to sending an A_ASSOC_REQ primitive.

An implementation capable of simultaneously supporting more than one client is capable of routing an incoming A_ASSOC_IND primitive to the particular APS Instance bound to the called presentation address. The API supports applications that are capable of receiving A_ASSOC_IND primitives addressed to a range of presentation addresses by means of *wildcard* presentation addresses. Two classes of wildcard addresses are supported: NSAP *wildcard* and *selector wildcard*. The two classes are discussed separately below; however, the service supports the use of both selector wildcards and NSAP wildcards within a single presentation address. Aspects of wildcard addresses relevant to both classes are discussed here.

3.3.8.1 Selector Wildcard Addresses

Different clients within a single computer system will likely share the same network address(es). Consequently, an APS API implementation is capable of differentiating between two unique presentation addresses based upon their distinct T-selector, S-selector or P-selector values, or any combination of the three. This specification does not impose any constraints on the strategy followed by an implementation to achieve unambiguous presentation addresses.

A *selector wildcard* presentation address is defined as a presentation address where one or both of the S-selector and P-selector components have not been specified. The unspecified component(s) are thus allowed to match any corresponding selector value(s) specified on the *called presentation address* field of an A_ASSOC_IND primitive.

NOTE — It should be noted that an unspecified selector value is distinct from a null selector value.

The unspecified components of a selector wildcard presentation address shall follow a bottom-to-top structure. Thus, if the T-selector component is not specified, both the S-selector and the P-selector cannot be specified. Likewise, if the S-selector is not specified, the P-selector cannot be specified.

The use of *selector wildcard* presentation addresses is restricted to applications that only support a responder role. They can only be used as values for the BIND_PADDR attribute and can only be set when the APS Instance is in the states UNBOUND or IDLE.

This specification does not impose any other constraints on how an implementation provides selector wildcard presentation addresses. Some implementations may provide selector wildcarding capabilities at both Session and Presentation layers, while others may not provide them at either layer.

A client bound to a selector wildcard presentation address is effectively bound to the lowest level SAP(s) for which an address has been specified. For example, a client bound to a wildcard address for which a T-selector value has been specified and the S- and P-selectors have been omitted is in fact bound to the Transport Layer SAP(s) identified by the specified transport address (consisting of the T-selector value and network address(es)).

Any incoming A_ASSOC_IND primitive addressed to the specified transport address is passed to the client bound to the wildcard address irrespective of the called S- and P-selector values. The client is then responsible for validating the called S- and P-selector values and determining whether or not to accept the application association.

Note that multiple selector wildcard addresses can freely coexist with each other and with fully specified addresses. The service always passes an incoming A_ASSOC_IND to the client bound to the most specific presentation address that matches the *called presentation address*.

3.3.8.2 NSAP Wildcard Addresses

Where a single computer system supports multiple network addresses, the service allows an application to listen for connection indications on one or more of these addresses by specifying a list of *NSAPs* as part of the presentation address assigned to BIND_PADDR. Some implementations may allow an application to specify an empty list to accept connections on any of the local *NSAPs*; this is termed an *NSAP wildcard* address.

NSAP wildcard addresses may be used by APS Instances supporting an initiator role, a responder role, or both. In the case where an APS Instance that is bound to an NSAP wildcard address sends an A_ASSOC_REQ primitive, the service provider determines which local NSAP to use to connect to the presentation address specified in REM_PADDR.

3.3.8.3 General Comments

When an indication is received by an APS Instance that is bound to a wildcard address, the environment variable LCL_PADDR is set to the specific presentation address to which the association is directed (the *called presentation address* from the A_ASSOC_REQ primitive). The address in LCL_PADDR is always used as the responding presentation address when sending an A_ASSOC_RSP primitive. The client may respond to an incoming A_ASSOC_IND on an address other than the called presentation address by specifying a fully *specified address* for BIND_PADDR and then invoking the `bind_address` operation, which in turn sets the value of LCL_PADDR. It should be noted that new values may be specified for the P-selector and S-selector only. An attempt to specify values for the T-selector or *NSAP* that differ from those received in LCL_PADDR results in the primitive being rejected by the service with an error code of `bind_refused`.

3.3.9 Memory Management Mechanisms

The service client may elect to handle all allocation and deallocation of memory for use by the service (this includes memory for APS environment attributes returned from the `get_attribute_value` operation, and buffers for receiving incoming user data). This mechanism enables the client to control the amount of buffer space used by the service for receiving incoming data.

To do this, the client registers client-supplied allocation and deallocation operations with the service when the `open_instance` or `restore_instance` operation is called. The service calls the client-supplied allocation routine in the following circumstances:

- If the client has set the *storage_allocation_mode* argument of the `open_instance` or `restore_instance` operation:
when an environment attribute value is to be returned to the client by the `get_attribute_value` operation, or primitive parameter values are to be returned by the `receive_primitive` or `look_at_primitive` operations, then the allocation operation is invoked with the *allocation_type* argument set to MEMORY.
- When user data is to be returned by the `receive_primitive` or `look_at_primitive` operation, and the user has not provided sufficient user data buffers and the *allocate_buffer_flag* argument was set to TRUE, then the allocation operation is called with the *allocation_type* argument set to BUFFERS.

The service invokes the client-supplied deallocation operation from the `free_memory` operation when the client passes any object that was allocated in the above circumstances.

The service is responsible for freeing memory that it allocated when it is no longer required. Exceptions to this are:

- Memory allocations passed to the client as part of the *received_parameters* argument of the `receive_primitive` or `look_at_primitive` operations
- Buffers passed to the client containing received user data
- Memory allocations passed to the client on return from the `get_attribute_value` operation

3.3.10 Primitive Parameters

The datatype `primitive_parameters_type` is used to pass the parameters associated with individual primitives between the client and the send primitive and receive primitive operations. The service maintains the parameters associated with the A_ASSOC primitives in the AP environment. The `primitive_parameters_type` datatype allows the environment values to be overridden when `send_primitive` is used to send an A_ASSOC_REQ or A_ASSOC_RSP primitive.

For **receive_primitive**, the datatype can be used to return some environment values directly, rather than requesting them using the **get_attribute_value** operation.

The **primitive_parameters_type** datatype is a collation of all possible parameters to all the primitives supported by the service. The specific use of the elements of the datatype depends upon the parameters defined for the specific service primitive being sent or received. Consequently, a description of how the associated operation argument is used for each of the primitives is included in section 6.

3.3.11 Error Reporting

Most operations return a status using the **result_type** datatype. The *result_type.result* element is set to SUCCESS if the operation was completed successfully.

The service reports internal error conditions caused, for example, by an invalid argument or a primitive issued out of state by setting *result_type.result* to AP_ERROR. In addition, several error *classes* have been identified to allow errors reported by an underlying service provider to be passed to the client. These are indicated by distinct values for the *result_type.result* element.

System errors that occur outside the scope of the APS API result in *result_type.result* to OS_ERROR, with error code being set to indicate the type of the system error.

To facilitate application portability, all implementations shall adhere to this scheme for reporting errors. However, only the errors belonging to the AP_ERROR class are mandatory. Specific errors belonging to the other error classes are dependent upon the underlying service providers utilized in a specific implementation. A complete list of error codes belonging to the AP_ERROR class reported through **result_type** is presented in 5.5.

An **error_text** operation is provided to return a text message that describes the error code passed to it.

3.3.12 Execution Mode

The **send_primitive** and **receive_primitive** operations may be used in either *blocking* or *non-blocking* execution mode.

In blocking mode, **send_primitive** blocks until resources are available to send the specified primitive in its entirety. Thus, if **send_primitive** is invoked when the communication path is flow controlled, the call blocks until the entire message has been sent. Similarly, in blocking mode, **receive_primitive** blocks until either an entire primitive is received, or the service fills the available buffering. In the latter case, the *more_data_flag* argument is set to TRUE when **receive_primitive** returns control to the client. In order to receive the remainder of the primitive, the client continues invoking **receive_primitive** until the operation returns with the *more_data_flag* argument set to FALSE.

NOTE — Either operation may be interrupted by an incoming primitive that causes the **look_at_inbound_primitive** error code to be returned.

When the service is used in non-blocking mode, **send_primitive** and **receive_primitive** never block. Hence, if **send_primitive** is called when insufficient resources are available to send the specified primitive in its entirety, the service provider accepts as much of the primitive as possible before returning with the error code **try_again**. To complete sending the primitive, the client calls **send_primitive** again with the same set of buffers as arguments until it returns successfully. The **poll_events** operation is provided to allow the client to wait until resources are available to send data on a particular APS Instance.

In the non-blocking mode, **receive_primitive** reads data from the instance until an entire primitive is received, or the available buffering is full, or no more data is available to be read, at which point **receive_primitive** returns control to the client. If the operation returns because no primitive is available (or no further data is available for a primitive of which part has already been returned), the **try_again** error code is returned. If the operation returns the status

SUCCESS, the *more_data_flag* argument may be checked to determine whether the current primitive has been completely received.

The *poll_events* operation can be used to wait until more data is available to be read from the instance.

NOTE — Again, either operation may be interrupted by an incoming primitive that causes the *look_at_inbound_primitive* error code to be returned.

The client can request either blocking or non-blocking execution through an argument to the *open_instance* operation, or by setting or resetting the *non_blocking* flag of the FLAGS environment attribute.

3.3.13 User Data Encoding

All *user* data passed between the service and the client is in encoded form, using one of the selected transfer syntaxes. The client is responsible for all user data encoding and decoding.

In addition, the client is responsible for generating the encoding that the ACSE and Presentation protocols require for including the user data in their PDUs.

NOTE — Conceptually, this is the task of the service provider; however, the APS API requires the client to do it for the sake of efficiency.

The descriptions of the *send_primitive* and *receive_primitive* operations in section 5 along with the descriptions of individual primitives given in section 6 describe how the user data shall be encoded for a particular primitive.

Two examples are given here of how information to be passed to the API shall be encoded. The examples make some assumptions about how the transfer syntax for the encoding is identified and about the choices that are made when information is passed to the ASCE and Presentation layers. The user is referred to the referenced specifications for the ACSE and Presentation protocols and for ASN.1 for precise information on this subject.

The examples use the *value syntax* defined by the ASN.1 specification with the addition (for the purposes of this discussion only) of explicit tag information included within square brackets (for example, [INTEGER] - a *universal tag*, [0] - a *context-specific tag*, and so on).

Example 1: An FTAM initiator implementation wishes to use the ACSE protocol to set up an application-association and transfer the F-CONNECT request primitive. It shall encode the following data and tags into the buffer passed as the *user_data_to_send* argument to the *send_primitive* operation that specifies A_ASSOC_REQ as the *primitive_to_send* argument.

```

user-information [30] {
  [EXTERNAL] {
    -- Association-information
    direct-reference [OBJECT IDENTIFIER] { . . . ftam-pci(1) }
    -- object id for the FTAM PDU abstract syntax
    single-ASN1-type [0] {
      f-initialize-request [0] {
        protocol-version [0] { version-1 } ,
        implementation-info [1] "ACME FTAM Initiator" ,
        .
        .
        .
      }
    }
  }
}

```

Example 2: The FTAM initiator subsequently wishes to transfer a series of strings from an unstructured text file. It shall encode the following data and tags into the buffer passed as the *user_data_to_send* argument to the *send_primitive* operation that specifies P_DATA_REQ as the *primitive_to_send* argument.

```
[APPLICATION 1] {      -- fully encoded user data
  [SEQUENCE] {        -- first PDV-List
    [INTEGER] <n>,      -- Identifies the abstract syntax
                      -- "FTAM unstructured text"
                      -- in the defined context set

    single-ASN1-type [0] {
      [GraphicString] "To begin at the beginning:"
                      -- first string from file
    }
  },
  [SEQUENCE] {        -- second PDV-List
    [INTEGER] <n>,      -- Identifies the abstract syntax
                      -- "FTAM unstructured text"
                      -- in the defined context set

    single-ASN1-type [0] {
      [GraphicString] "It is spring, moonless night in the
                      small town, starless and Bible-black..."
                      -- second string from file
    }
  },
  .
  .
}
```

Example 3: The following encoding is an alternative to that shown in example 2.

```
[APPLICATION 1] {  -- fully encoded user data
  [SEQUENCE] {  -- first PDV-List
    [INTEGER] <n> ,  -- Identifies the abstract syntax
                  -- "FTAM unstructured text"
                  -- in the defined context set
    octet-aligned [1] {
      [GraphicString] "To begin at the beginning:"
                  -- first string from file
      [GraphicString] "It is spring, moonless night in the small
town, starless and bible-black..."
                  -- second string from file
    }
  },
  .
  .
}
```

3.3.14 More_data_flag Argument

The client may choose to send the encoded data associated with a particular primitive as a series of *send_primitive* invocations. This is accomplished by setting the *more_data_flag* argument to the *send_primitive* operation to TRUE. The user may continue sending data associated with the current primitive by repeatedly invoking *send_primitive* with the *more_data_flag* argument set to TRUE. When the last buffer of data associated with the primitive is sent, *send_primitive* shall be invoked with the *more_data_flag* argument set to FALSE. This final invocation in a sequence shall contain at least one octet of user data.

NOTE — Where an indeterminate amount of user data is being sent with a primitive, (using a sequence of `send_primitive` invocations) the application must ensure that the data is passed to the service in such a way that this requirement is satisfied.

The *primitive_to_send* argument shall remain the same for each subsequent invocation until the primitive is complete. The *parameters_to_send* argument also should not change; however, since this argument is only examined on the first `send_primitive` invocation in such a sequence, no error is reported if its value is subsequently modified.

If all data associated with the primitive is sent on the first `send_primitive` invocation, the operation shall be issued with the *more_data_flag* argument set to FALSE.

In a similar fashion, if the service returns data associated with a received primitive via multiple invocations of the `receive_primitive` operation, it sets the *more_data_flag* output argument to TRUE. The `receive_primitive` sets the *more_data_flag* output argument to FALSE when the received primitive is complete.

NOTE — It should be noted that the value of the *received_primitive* argument should be checked on return from each `receive_primitive` call because an interrupting primitive (for example, `A_PABORT_IND`) could arrive before all data associated with the first primitive is processed.

3.4 Using the APS API

Below is a summary of the steps required to establish an association with a remote application entity using the APS API. The procedure presented is intended solely as a general description of how the interface might be used. It should not be construed as an attempt to provide a template for constructing any particular application. Moreover, it is assumed that the client is familiar with the ACSE and Presentation Layer protocols and understands the role of the ACSE-service-user in establishing, using, and terminating an association between two application entities.

- 1) *Obtain an APS Instance.* First, a APS Instance is created. This is accomplished either by using `open_instance`, or by acquiring an already established instance through some other implementation-specific mechanism.
- 2) *Initialize the Environment.* Next, the environment is initialized using the `init_environment` operation. After the environment is initialized, the user may examine or alter the environment attributes, subject to the readability and writability restrictions discussed in section 4.
- 3) *Bind the APS Instance to a Presentation Address.* Before any service primitives can be sent or received, the APS Instance shall be bound to a presentation address. This is accomplished by setting the `BIND_PADDR` attribute. The `BIND_PADDR` attribute can be set using either the `set_attribute_value` operation or the `init_environment` operation. Then the service is instructed to activate the APS Instance by using the `bind_address` operation.
- 4) *Set Other Environment Attributes.* In addition to `BIND_PADDR`, other environment variables shall be set before an association is established, particularly if the client is the association-initiator. For example, before issuing an `A_ASSOC_REQ` primitive, the client specifies the address of the remote entity by setting the `REM_PADDR` attribute and set the `PCDL` attribute to propose a list of presentation contexts that are to be used for transferring user data.
Refer to section 4 and the tables in section 6 for further information about the environment attributes and how they relate to sending and receiving individual primitives.
- 5) *Send or Receive A_ASSOC Service Primitives.* Once the environment of an APS Instance has been properly initialized, the client may attempt to establish an association with a remote application entity by using the `A_ASSOC` primitives in conjunction with the `send_primitive` and `receive_primitive` operations.
- 6) *Transfer Data.* Once the association is established, information may be exchanged with the remote application entity by sending and receiving the appropriate data transfer primitives (using `send_primitive` and `receive_primitive`).
- 7) *Release the Association.* An association may be released either normally (using the `A_RELEASE_REQ` primitive), or abnormally (by either the `A_ABORT` or the `A_PABORT` primitives).

4. APS Environment

The ACSE and Presentation Layer API environment is made up of a set of attributes that are used to keep state information and to store data required to establish and maintain an association with another application entity. Clause 4.1 describes the meaning and use of the environment attributes that comprise the API environment. Clause 4.2 summarizes the relationships between the environment attributes (where setting one attribute value affects the value of another). Clause 4.3 presents the attributes in tabular form, with information about the states in which an attribute is readable and writable, the associated datatype, its possible values, and the the default value where appropriate.

4.1 API Attribute Descriptions

For each attribute, the following information is listed:

- The meaning and use of the attribute
- Where appropriate, its mapping to ACSE/Presentation service parameters
- Its relationship to other attributes (where setting its value may affect another attribute or where its value is affected by setting another attribute)
- Whether the attribute applies to the X.410-1984 mode of operation. (Refer to the description of `MODE_SEL` in 4.1.29)
- The LI datatype used to get or set its value

4.1.1 ACSE_AVAIL

This attribute indicates which versions of the ACSE protocol are currently available.

This attribute is not used in the X.410-1984 mode.

The LI datatype `acse_version_type` is used to get or set the value of this attribute.

4.1.2 ACSE_SEL

This attribute indicates which version of the ACSE protocol has been selected for use with the current association.

This attribute is not used in the X.410-1984 mode.

The LI datatype `acse_version_type` is used to get or set the value of this attribute.

4.1.3 AFU_AVAIL

This attribute indicates which optional ACSE functional units are currently available.

This attribute is not used in the X.410-1984 mode.

The LI datatype `afu_type` is used to get or set the value of this attribute.

4.1.4 AFU_SEL

This attribute indicates which optional ACSE functional units have been requested for use over the current association.

This attribute is not used in the X.410-1984 mode.

The LI datatype `afu_type` is used to get or set the value of this attribute.

4.1.5 BIND_PADDR

This attribute is used to indicate the local presentation address to which this APS Instance shall be bound when the `bind_address` operation is invoked.

Setting this attribute shall result in the attribute `LCL_PADDR` being set to the same value.

NOTE — The attributes `BIND_PADDR` and `LCL_PADDR` do sometimes have different values. For example, a *listener* application might set the `BIND_PADDR` to a *wildcard* address and use the called address (indicated by `LCL_PADDR`) specified in each indication primitive to dispatch associations.

The LI datatype `paddr_type` is used to get or set the value of this attribute.

4.1.6 CLD_AEID

This attribute is the *called AE invocation-identifier* parameter of the `A_ASSOC_REQ` and `A_ASSOC_IND` primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aei_api_id` type is used to get or set the value of this attribute.

4.1.7 CLD_AEQ

This attribute is the *called AE-qualifier* parameter of the `A_ASSOC_REQ` and `A_ASSOC_IND` primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aeq_type` is used to get or set the value of this attribute.

4.1.8 CLD_APIID

This attribute is the *called AP invocation-identifier* parameter of the `A_ASSOC_REQ` and `A_ASSOC_IND` primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aei_api_id_type` is used to get or set the value of this attribute.

4.1.9 CLD_APT

This attribute is the *called AP-title* parameter of the `A_ASSOC_REQ` and `A_ASSOC_IND` primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `apt_type` is used to get or set the value of this attribute.

4.1.10 CLD_CONN_ID

This attribute is the value of the *session connection identifier* that was proposed by the association-responder.

The LI datatype `conn_id_type` is used to get or set the value of this attribute.

4.1.11 CLG_AEID

This attribute is the *calling AE invocation-identifier* parameter of the A_ASSOC_REQ and A_ASSOC_IND primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aei_api_id_type` is used to get or set the value of this attribute.

4.1.12 CLG_AEQ

This attribute is the *calling AE-qualifier* parameter of the A_ASSOC_REQ and A_ASSOC_IND primitives.

See the description of `aeq_type` in 5.2.4.1 for information about how its absence is indicated.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aeq_type` is used to get or set the value of this attribute.

4.1.13 CLG_APIID

This attribute is the *calling AP invocation-identifier* parameter of the A_ASSOC_REQ and A_ASSOC_IND primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aei_api_id_type` is used to get or set the value of this attribute.

4.1.14 CLG_APT

This attribute is the *calling AP-title* parameter of the A_ASSOC_REQ and A_ASSOC_IND primitives.

The LI datatype `apt_type` is used to get or set the value of this attribute.

See the description of `apt_type` in 5.2.5.1 for information about how its absence is indicated.

4.1.15 CLG_CONN_ID

This attribute is the value of the *session connection identifier* that was proposed by the association-initiator.

The LI datatype `conn_id_type` is used to get or set the value of this attribute.

4.1.16 CNTX_NAME

This attribute is the *application context name* parameter of the A_ASSOC_REQ, A_ASSOC_IND, A_ASSOC_RSP, and A_ASSOC_CNF primitives.

The LI datatype `objid_type` is used to get or set the value of this attribute.

See the description of `objid_type` in 5.2.19.1 for information about how its absence is indicated.

This attribute is not used in the X.410-1984 mode.

4.1.17 COPYENV

If this attribute is set to TRUE, the service shall return the values of certain environment attributes that correspond to parameters on the A_ASSOC_IND and A_ASSOC_CNF primitives in the *received_parameters* argument of the *receive_primitive* operation. When FALSE, these attributes shall not be returned.

4.1.18 DCS

This attribute (the *defined context set*) consists of a list of abstract syntax/transfer syntax pairs that were negotiated when the association was established, together with the presentation context identifier that identifies the pair.

The value of DCS shall be empty (containing no entries) in the states IDLE and WASSOCcnf_ASSOCreq, it shall contain the ACSE context negotiated by the APS Instance in the state WASSOCrsp_ASSOCind, and it shall contain the fully negotiated defined context set in all other states.

The defined context set shall contain both the presentation contexts that are negotiated by the client using PCDL and PDCR and any additional presentation contexts that are negotiated by the AP Instance or the service provider. For example, the ACSE context, {joint-iso-ccitt 2 1 0 1}, is negotiated automatically and shall appear in the defined context set after negotiation is complete.

This attribute is not used in the X.410-1984 mode.

The LI datatype *dcstype* is used to get or set the value of this attribute.

4.1.19 DIAGNOSTIC

This attribute is used to pass diagnostic details on error conditions reported by ACSE, presentation, session, and transport providers, in addition to that conveyed with the A_PABORT_IND primitive. This mechanism provides descriptive text for all errors reported by service, and may be used to provide additional diagnostic information (for example, errors in incoming APDUs reported by the local ACSE protocol machine, or the reason for a transport layer disconnect).

The LI datatype *diag_type* is used to get or set the value of this attribute. (See 5.2.13.1 for a list of the possible values for this attribute).

The information provided by this attribute is implementation-dependent. That is, an implementation may not report all of the source/reason/event combinations defined for this attribute. However, if an implementation does use the attribute to report diagnostic information, it shall use the classifications defined here.

4.1.20 DPCN

This attribute is the *default presentation context name* parameter of the A_ASSOC_REQ and A_ASSOC_IND primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype *dcn_type* is used to get or set the value of this attribute.

4.1.21 DPCR

This attribute is the *default presentation context result* parameter of the A_ASSOC_CNF primitive. Since the client is responsible for encoding and decoding of user data, it is the responsibility of the client to accept or reject any proposed default presentation context. The value DPCR_NOVAL indicates that it is not present.

This attribute is not used in the X.410-1984 mode.

The LI datatype `dpcr_type` is used to get or set the value of this attribute.

4.1.22 FLAGS

This attribute consists of an array of Boolean flags used to control the characteristics of this APS Instance. There is currently only one flag defined:

`non_blocking`

If this flag is set, the APS Instance operates in non-blocking execution mode.

The LI datatype `flags_type` is used to get or set the value of this attribute.

4.1.23 INIT_SYNC_PT

This attribute is the *initial synchronization point serial number* parameter of the A_ASSOC_REQ, A_ASSOC_IND, A_ASSOC_RSP, and A_ASSOC_CNF primitives.

The LI datatype `init_sync_pt_type` is used to get or set the value of this attribute.

4.1.24 INIT_TOKENS

This attribute is the *initial assignment of tokens* parameter of the A_ASSOC_REQ, A_ASSOC_IND, A_ASSOC_RSP, and A_ASSOC_CNF primitives.

This attribute represents the proposed initial assignment of the tokens defined by the session protocol. It consists of an entry for each token:

- Release token
- Major/activity token
- Synchronize-minor token
- Data token

Each entry identifies the initial assignment proposed for the token:

- Initiator's side
- Responder's side
- Called SS user's choice

Tokens left as “called SS user's choice” by the association responder in the A_ASSOC_RSP primitive shall be assigned to the initiator by default.

The service shall ignore assignments for tokens that are unavailable due to the absence of support for the associated session functional unit.

The LI datatype `init_tokens_type` is used to get or set the value of this attribute.

4.1.25 LCL_PADDR

This attribute is the presentation address of the local entity. If the local entity is the initiator of an association, this value shall be used as the *calling presentation address* of the A_ASSOC_REQ, primitive. If the local entity is the association-responder, this value is the *called presentation address* parameter received in the A_ASSOC_IND primitive, and shall be used as the *responding presentation address* when the A_ASSOC_RSP primitive is issued.

This attribute is not directly settable by the client. However, setting BIND_PADDR shall result in the LCL_PADDR attribute being set to the same value.

NOTE — The attributes BIND_PADDR and LCL_PADDR do not always have the same value. See 4.1.5.

The LI datatype `paddr_type` is used to get or set the value of this attribute.

4.1.26 LIB_AVAIL

This attribute indicates which versions of the APS API are supported by this implementation of the APS API.

The LI datatype `lib_version_type` is used to get or set the value of this attribute.

4.1.27 LIB_SEL

This attribute is used to indicate which version of the APS API is in use. To facilitate application portability, this attribute should be set before any other attributes are set or any primitives are issued.

NOTE — This attribute has no default value. When no value is assigned to this attribute the effect of setting the value of any other attribute, or of calling any XAP function (other than `set_attribute_value`) is unspecified.

The LI datatype `lib_version_type` is used to get or set the value of this attribute.

4.1.28 MODE_AVAIL

This attribute indicates the available modes of operation for the APS Instance and service provider. The modes that may be supported are *normal* and *X.410-1984*.

The LI datatype `mode_type` is used to get or set the value of this attribute.

4.1.29 MODE_SEL

This attribute indicates the mode of operation (*normal* or *X.410-1984*) that has been selected. This value is used as the *mode* parameter of the A_ASSOC_REQ and A_ASSOC_IND primitives.

The LI datatype `mode_type` is used to get or set the value of this attribute.

When this attribute is set to X410_MODE (indicating the X.410-1984 mode is in effect), certain other attributes are not used. The readability and writability of these attributes is not affected but their values shall not be used or updated by the service. A list of the attributes that are not used in the X.410-1984 mode is provided in Table 4-1.

Table 4-1 — Attributes NOT Used in X.410-1984 Mode

ACSE_AVAIL	CLG_AEID	PCDL	RSP_AEQ
ACSE_SEL	CLG_AEQ	PCDRL	RSP_APIID
CNTX_NAME	CLG_APIID	PFU_AVAIL	RSP_APT
CLD_AEID	CLG_APT	PFU_SEL	
CLD_AEQ	DPCN	PRES_AVAIL	
CLD_APIID	DPCR	PRES_SEL	
CLD_APT	DCS	RSP_AEID	

4.1.30 MSTATE

This attribute consists of a set of Boolean flags that indicate the current state of an APS Instance. The following flags are defined:

send_more

This flag is set if the APS Instance is awaiting additional data from the client (the preceeding **send_primitive** operation had the *more_data_flag* argument set to TRUE).

receive_more

This flag is set if there is more user data for the current service (the last **receive_primitive** operation returned with the *more_data_flag* argument set to TRUE).

NOTE — It is possible for both flags to be set simultaneously.

This attribute is not used in the X.410-1984 mode.

The LI datatype **restate_type** is used to get or set the value of this attribute.

4.1.31 OLD_CONN_ID

This attribute is the value of the session connection identifiers from a previous session.

The LI datatype **old_conn_id_type** is used to get or set the value of this attribute.

4.1.32 OPT_AVAIL

This attribute consists of a set of Boolean flags that indicate the optional functionality that is supported in the underlying protocol implementation. The following flags are defined:

nsap_wild

If set, indicates that wildcarding is supported at the NSAP level.

tsel_wild

If set, indicates that wildcarding is supported at the Transport Selector level.

ssel_wild

If set, indicates that wildcarding is supported at the Session Selector level.

psel_wild

If set, indicates that wildcarding is supported at the Presentation Selector level.

The LI datatype **opt_avail_type** is used to get or set the value of this attribute.

4.1.33 PCDL

This attribute is used as the *presentation context definition list* parameter of the A_ASSOC_REQ and A_ASSOC_IND primitives.

The association-initiator sets this list to indicate the contexts it wishes to use for encoding user data on the association. Since a client is responsible for encoding and decoding all user data, a client acting as the association-initiator shall include the proposed transfer syntaxes for each proposed abstract syntax in the list submitted. For the association-responder, the list indicates which presentation contexts are being requested by the association-initiator for use on the association.

A client acting as the association-initiator may specify the ACSE context in PCDL although it is not required to do so. If the initiating client does not include the ACSE context in PCDL, the service shall add it. Regardless of whether or not the initiating client includes the ACSE context, it shall not be indicated in PDCL to a client acting as the association-responder. A responding client may obtain it from the DCS attribute if required.

This attribute is not used in the X.410-1984 mode.

The LI datatype `cdl_type` is used to get or set the value of this attribute.

4.1.34 PCDRL

This attribute is the *presentation context definition result list* parameter of the A_ASSOC_IND, A_ASSOC_RSP, and A_ASSOC_CNF primitives.

It is used to indicate whether the presentation contexts proposed in the *presentation context definition list* have been accepted or rejected. A client acting as the association-responder uses this attribute to indicate which of the proposed presentation contexts are acceptable when sending the A_ASSOC_RSP primitive. For a client acting as the association-initiator, this attribute indicates the response of the association-responder, received in the A_ASSOC_CNF primitive, to the presentation contexts proposed in a preceeding A_ASSOC_REQ primitive.

Each entry in PCDRL corresponds to an entry in PCDL. In the case of a responding client, PCDRL never contains the proposed ACSE context. A responding client has an opportunity to accept or reject each of the proposed contexts that are indicated in PCDL. The ACSE context is automatically accepted by the service and appears only in the value of DCS.

This attribute is not used in the X.410-1984 mode.

The LI datatype `cdrl_type` is used to get or set the value of this attribute.

4.1.35 PFU_AVAIL

This attribute indicates which optional Presentation Layer functional units are currently available.

This attribute is not used in the X.410-1984 mode.

The LI datatype `pfu_type` is used to get or set the value of this attribute.

4.1.36 PFU_SEL

This attribute is the *Presentation requirements* parameter of the A_ASSOC_REQ, A_ASSOC_IND, A_ASSOC_RSP, and A_ASSOC_CNF primitives.

It indicates which optional Presentation Layer functional units have been requested or selected for use over the current association.

This attribute is not used in the X.410-1984 mode.

The LI datatype `pfu_type` is used to get or set the value of this attribute.

4.1.37 PRES_AVAIL

This attribute indicates which versions of the Presentation Layer protocol are currently available.

The LI datatype `pres_version_type` is used to get or set the value of this attribute.

4.1.38 PRES_SEL

This attribute indicates which version of the Presentation Layer protocol has been selected for use with the current association.

This attribute is not used in the X.410-1984 mode.

The LI datatype **pres_version_type** is used to get or set the value of this attribute.

4.1.39 QLEN

This attribute is used to indicate the number of connection indications that will be queued by the service when all of the connection endpoints bound for listening at the presentation address identified by **BIND_PADDR** are in use. The client can set this attribute to request a specific queue size. The provider may change the value if it cannot support the requested queue size or if another process bound to the same address has already requested a larger queue. The client can determine the current queue size by examining the value of **QLEN**.

The LI datatype **integer** is used to get or set the value of this attribute.

4.1.40 QOS

This attribute is used as the *quality of service* parameter of the **A_ASSOC_REQ**, **A_ASSOC_IND**, **A_ASSOC_RSP**, and **A_ASSOC_CNF** primitives.

It specifies the quality of service requirements for the association. It holds the ranges of **QOS** values that the client is willing to accept.

The LI datatype **qos_type** is used to get or set the value of this attribute.

4.1.41 REM_PADDR

This attribute holds the value used as the presentation address of the remote entity. If the local entity is the initiator of an association, this value is the *called presentation address* parameter of the **A_ASSOC_REQ** primitive. If the local entity is the association-responder, this value is the *calling presentation address* parameter of the **A_ASSOC_IND** primitive.

The LI datatype **paddr_type** is used to get or set the value of this attribute.

4.1.42 ROLE_ALLOWED

This attribute specifies the association roles that APS Instance may perform. It consists of a pair of Boolean flags that indicate which roles the APS Instance supports. The instance may be used as an association-initiator if *initiator*=TRUE (indicates that the client may send an **A_ASSOC_REQ** primitive). The instance may be used as an association-responder if *responder*=TRUE (indicates that the client may receive an **A_ASSOC_IND** primitive). Both flags may be set if both roles are supported.

This attribute only affects the roles that an APS Instance may play with respect to association establishment. Changing the value of this attribute will not affect the way in which the APS Instance participates in an association that has already been established.

NOTE — If an **A_ASSOC_IND** primitive has been queued prior to setting **ROLE_ALLOWED** to **INITIATOR**, an **A_ASSOC_IND** primitive may be received even after *responder* has been set to **FALSE** to prohibit receipt of association indications. To ensure that this does not happen, the client should set the **ROLE_ALLOWED** attribute before the APS Instance is bound to a presentation address.

The LI datatype **role_type** is used to get or set the value of this attribute.

4.1.43 ROLE_CURRENT

This attribute indicates the role of the client in the current association. The attribute is set to INITIATOR as soon as an A_ASSOC_REQ primitive is sent and is cleared when the association is rejected or subsequently terminated. Similarly, the attribute is set to RESPONDER upon receipt of an A_ASSOC_IND and is cleared when the association is terminated.

The LI datatype `role_type` is used to get or set the value of this attribute.

4.1.44 RSP_AEID

This attribute is the *responding AE invocation-identifier* parameter of the A_ASSOC_RSP and A_ASSOC_CNF primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aei_api_id_type` is used to get or set the value of this attribute.

4.1.45 RSP_AEQ

This attribute is the *responding AE qualifier* parameter of the A_ASSOC_RSP and A_ASSOC_CNF primitives.

See the description of `aeq_type` in 5.2.4.1 for information about how its absence is indicated.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aeq_type` is used to get or set the value of this attribute.

4.1.46 RSP_APIID

This attribute is the *responding AP invocation-identifier* parameter of the A_ASSOC_RSP and A_ASSOC_CNF primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `aei_api_id_type` is used to get or set the value of this attribute.

4.1.47 RSP_APT

This attribute is the *responding AP-title* parameter of the A_ASSOC_RSP and A_ASSOC_CNF primitives.

This attribute is not used in the X.410-1984 mode.

The LI datatype `apt_type` is used to get or set the value of this attribute.

4.1.48 SESS_AVAIL

This attribute indicates which versions of the Session Layer protocol are currently available.

The LI datatype `sess_version_type` is used to get or set the value of this attribute.

4.1.49 SESS_SEL

This attribute indicates which version of the Session Layer protocol has been selected for use with the current association.

The LI datatype `sess_version_type` is used to get or set the value of this attribute.

4.1.50 SESS_OPT_AVAIL

This attribute is an array of Boolean flags used to indicate which optional session capabilities are supported by the service provider. The following flags are defined:

- `ucbc` The underlying provider supports client control of basic concatenation.
- `ucec` The underlying provider supports client control of extended concatenation.

For further information see the description of the `send_primitive` operation in 5.3.15.

The LI datatype `sess_options_type` is used to get or set the value of this attribute.

4.1.51 SFU_AVAIL

This attribute indicates which Session Layer functional units are currently available.

The value of this attribute may affect the value of `TOKENS_AVAIL`.

The LI datatype `sfu_type` is used to get or set the value of this attribute.

4.1.52 SFU_SEL

This attribute is the *session requirements* parameter of the `A_ASSOC_REQ`, `A_ASSOC_IND`, `A_ASSOC_RSP`, and `A_ASSOC_CNF` primitives. This attribute indicates which Session Layer functional units have been requested/selected for use over the current association.

The LI datatype `sfu_type` is used to get or set the value of this attribute.

4.1.53 STATE

This attribute is used to convey state information about the APS Instance. It is used to determine which primitives are legal, which attributes can be read/written, etc.

Table 4-3 contains information about how accessibility of environment attributes is affected by the value of this environment attribute. Table 6-5 and Table 6-7 show how the value of this primitive is affected by sending and receiving primitives.

The LI datatype `state_type` is used to get or set the value of this attribute.

4.1.54 TOKENS_AVAIL

A token is an attribute of an association that is dynamically assigned to one user at a time. The user that possesses a token has exclusive rights to initiate the service which that token represents. The `TOKENS_AVAIL` attribute indicates which tokens are available for assignment on this association.

The value of this attribute may be affected by the value of `SFU_SEL`.

The LI datatype `tokens_type` is used to get or set the value of this attribute.

4.1.55 TOKENS_OWNED

The TOKENS_OWNED attribute indicates which available tokens are currently assigned to the user. The user has exclusive rights to initiate the service represented by each of the tokens owned.

The LI datatype `tokens_type` is used to get or set the value of this attribute.

4.2 Environment Attribute Relationships

Table 4-2 shows the relationships between the environment attributes. The table lists each attribute that is party to a dependency relationship, identifying which values are affected by setting its value, and which attributes it is affected by.

Table 4-2—Environment Attributes—Relationships

Attribute Name	Affects	Is Affected By
BIND_PADDR	LCL_PADDR	
LCL_PADDR		BIND_PADDR
LIB_SEL	potentially all	
SFU_SEL	TOKENS_AVAIL	
STATE		BIND_PADDR
TOKENS_AVAIL		SFU_SEL

4.3 Environment Attribute Summary

Table 4-3 lists all the AP environment attributes, with the following columns:

Name	The symbolic name that is used to identify the attribute.
Type/Value	The datatype used to represent the attribute, with a list of possible values for the attribute and default value supplied by the service (if any). If no default value is supplied, the default value is given as “none” or “not present”. “None” implies that a value shall be specified by the client prior to invoking <code>send_primitive</code> , whereas “not present” implies that a value need not be specified, as the attribute represents an optional field of a particular primitive. They are identical otherwise.
Readable	The states in which the attribute may be read using <code>get_attribute_value</code> (states are given as values of the STATE environment attribute).
Writable	The states during which the attribute may be assigned a value using either <code>set_attribute_value</code> or <code>init_environment</code> (states are given as values of the STATE environment attribute).

Table 4.3—Environment Attributes—Summary

Attribute	Type/Values	Readable	Writable
ACSE_AVAIL	acse_version_type boolean flags: acsever1	always	never
ACSE_SEL	acse_version_type boolean flags: acsever1 default: acsever1=TRUE	always	only in states: UNBOUND IDLE
AFU_AVAIL	afu_type boolean flags: none	always	never
AFU_SEL	afu_type boolean flags: none default: none	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
BIND_PADDR	paddr_type default: none	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind

Table 4.3 — Environment Attributes—Summary (continued)

Attribute	Type/Values	Readable	Writable
CLD_AEID	aei_api_id_type default: not present	always	only in states: UNBOUND IDLE
CLD_AEQ	aeq_type default: not present	always	only in states: UNBOUND IDLE
CLD_APIID	aei_api_id_type default: not present	always	only in states: UNBOUND IDLE
CLD_APT	apt_type default: not present	always	only in states: UNBOUND IDLE
CLD_CONN_ID	conn_id_type default: not present	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
CLG_AEID	aei_api_id_type default: not present	always	only in states: UNBOUND IDLE
CLG_AEQ	aeq_type default: not present	always	only in states: UNBOUND IDLE
CLG_APIID	aei_api_id_type default: not present	always	only in states: UNBOUND IDLE
CLG_APT	apt_type default: not present	always	only in states: UNBOUND IDLE
CLG_CONN_ID	conn_id_type default: not present	always	only in states: UNBOUND IDLE
COPYENV	boolean default: FALSE	always	always
CNTX_NAME	objid_type default: none	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind

Table 4.3 — Environment Attributes—Summary (continued)

Attribute	Type/Values	Readable	Writable
DCS	dcs_type	in any states but: UNBOUND	never
DIAGNOSTIC	diag_type	always	never
DPCN	dcn_type default: not present	always	only in states: UNBOUND IDLE
DPCR	dpcr_type state values: DPCR_NOVAL ACCEPT USER_REJ PROV_REJ default: DPCR_NOVAL	only in state: WASSOCrsp_ASSOCind	only in state(s): WASSOrsp_ASSOCind
FLAGS	flags_type boolean flags: non_blocking	always	always
INIT_SYNC_PT	init_sync_pt_type default: MIN_SYNCP	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
INIT_TOKENS	init_tokens_type default: DATA_TOK_REQ, SYNCMINOR_TOK_REQ, MAJACT_TOK_REQ, RELEASE_TOK_REQ	only in states: UNBOUND IDLE WASSOCcnf_ASSOCreq WASSOCrsp_ASSOCind	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
LCL_PADDR	paddr_type	always	never
LIB_AVAIL	lib_version_type boolean flags: libver1	always	never
LIB_SEL	lib_version._type boolean flags: libver1 default: none	always	only in state: UNBOUND
MODE_AVAIL	mode_type boolean flags: normal_mode x410_mode	always	never

Table 4.3 — Environment Attributes—Summary (continued)

Attribute	Type/Values	Readable	Writable
MODE_SEL	mode_type boolean flags: normal_mode x410_mode default: normal_mode=TRUE x410_mode=FALSE	always	only in states: UNBOUND IDLE
MSTATE	mstate_type boolean flags: sndmore rcvmore	always	never
OLD_CONN_ID	old_conn_id_type default: not present	always	only in states: UNBOUND IDLE
OPT_AVAIL	opt_avail_type boolean flags: nswild tsel_wild ssel_wild psel_wild	always	never
PCDL	cdl_type default: not present	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind	only in states: UNBOUND IDLE
PCDRL	cdrl_type default: not present	only in states: WASOCrsp_ASSOCind DATA_XFER	only in states: WASSOCrsp_ASSOCind
PFU_AVAIL	pfu_type boolean flags: none	always	never
PFU_SEL	pfu_type boolean flags: none default: none	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
PRES_AVAIL	pres_version_type boolean flags: presver1	always	never

Table 4.3 — Environment Attributes—Summary (continued)

Attribute	Type/Values	Readable	Writable
PRES_SEL	pres_version_type boolean flags: presver1 default: presver1=TRUE	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
QLEN	integer{0,(impl def)} default: 0	always	only in state: UNBOUND
QOS	qos_type	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
REM_PADDR	paddr_type default: none	always	only in states: UNBOUND IDLE
ROLE_ALLOWED	role._type boolean flags: initiator responder default: initiator=TRUE responder=TRUE	always	always
ROLE_CURRENT	role_type boolean flags: initiator responder	in any states but: UNBOUND IDLE	never
RSP_AEID	aci_api_id_type default: not present	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
RSP_AEQ	aeq_type default: not present	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
RSP_APIID	aci_api_id_type default: not present	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind
RSP_APT	apt_type default: not present	always	only in states: UNBOUND IDLE WASSOCrsp_ASSOCind