The ATM Forum
Technical Committee

Native ATM Services: Semantic Description Version 1.0

af-saa-0048.000

February, 1996

The ATM Forum wishes to especially thank the following individuals who contributed much effort to the production of this specification:

Jim Harford	Chairman of the API Ad-hoc Work Group and directly responsible for most of the text herein
Bob Callaghan,	Editor of this specification

Dean Skidmore
Chairman, ATM Forum SAA Work Group

This page is intended to be blank.

Contents

1. Introduction

1.1 Purpose of Document

This document specifies the semantic definition of ATM-specific services that are available to software programs and hardware residing in devices on the user side of the ATM User-Network Interface. The ATM environment provides a wealth of new services to the application developer. This allows enhancements in performance and specification of network characteristics. Such ATM-specific services are denoted by the term "Native ATM Services".

"Semantic", means that this document will describe the services in a way that is independent of any programming language or operating system environment. Semantic specifications for generic services define various aspects of the interface to those underlying services, such as:

- request for the underlying service to perform some action
- notification that some event has occurred
- parameters of the requests and notifications
- response codes to the requests and notifications.

This document uses a state machine, entity-relation diagrams, primitives, implementation-specific tips, and other informative text to accomplish this aim.

The semantic description presented in this document is not an Applications Programming Interface (API). An API is a set of libraries or interfaces that enable an application to use the language in which it is written to access the functionality of lower-level modules - such as operating systems, graphical user interfaces, and communications protocols. The ATM Forum's interest is in applications being able to access the functionality provided by Native ATM Services.

This document will advance the development of such APIs that allow access to Native ATM Services. The semantic description included herein is intended to influence the direction of these emerging APIs. This document addresses concerns with both interoperability aspects and the proper abstraction of ATM procedures and parameters. Thus, this semantic description provides a firm engineering foundation and logically precedes any API development. Note that API development is expected to occur within the ATM Forum, other industry groups, and ATM vendors.

Note also that the semantic description herein is not limited to development of traditional APIs. Rather, this interface can be applied to any software program or hardware that uses Native ATM Services. Some examples of this are:

1. operating system kernel interface between Native ATM Services and ATM LAN Emulation
2. operating system kernel interface between Native ATM Services and traditional networking protocols (e.g. IP, X.25)
3. operating system kernel interface to an ATM device driver, where the device driver contains the implementation of the Native ATM Services
4. inside a PBX, used for circuit emulation
5. inside an ATM switch, used for vendor applications providing value-added services.

1.2 Scope of Document

"Native ATM Services" include the following:

1. data transfer, including both reliable and unreliable data delivery, using the ATM layer and various ATM adaptation layers

2. provisions for setting up switched virtual circuits (SVC)

3. provisions for setting up permanent virtual circuits (PVC)

4. traffic management considerations, including traffic types and quality of service guarantees

5. Distribution of connections and associated data to the correct application, or entity.

6. provisions for local participation in network management (i.e. ILMI and OAM protocols)

The next section presents a reference model showing the relationships between the various software components in a device at the user side of the ATM User-Network Interface. In the reference model, the dashed line labeled "Native ATM SAP" identifies the general scope and interest of this document.

This version of the specification supports version 3.x (3.0 and 3.1) of the ATM Forum's User-Network Interface (UNI) Specification. Future versions of this specification will support future versions of the UNI. Not all features of UNI 3.x are supported by this document. In particular, several limitations are listed below.

- UNI 3.x allows for two levels of ATM bearer service: virtual path (VP) and virtual channel (VC). This document supports only VC-level service. The support of VP-level service is for further study.

- UNI 3.x allows for AAL type 1, AAL type 3/4, AAL type 5, and a user-defined AAL. This document supports only AAL Type 1, AAL type 5, and a user-defined AAL. The support for AAL1 and user-defined AAL includes the control plane (signaling), but the data plane is considered implementation-specific at this time. The support of the other AAL types (including AAL type 2) is for further study.

- This document supports only the message mode of AAL type 5. Support of AAL type 5 streaming mode is for further study.

- The procedures in this document do not support a single ATM device being multiple leaves on a same point-to-multipoint connection. This feature is for further study.

- Some features of UNI 3.0 were changed or obsoleted by UNI 3.1. Where such conflicts exist, this document is consistent with UNI 3.1.

Motivation for a software interface specification centers on portability of software and reduction of development expense. In addition, this document addresses end-to-end interoperability issues, such as:

- distribution of an incoming connection to the correct application inside the device on the user side of the UNI

- agreement of what parameters supported in the UNI may be expected by the application

1.3 Reference Model

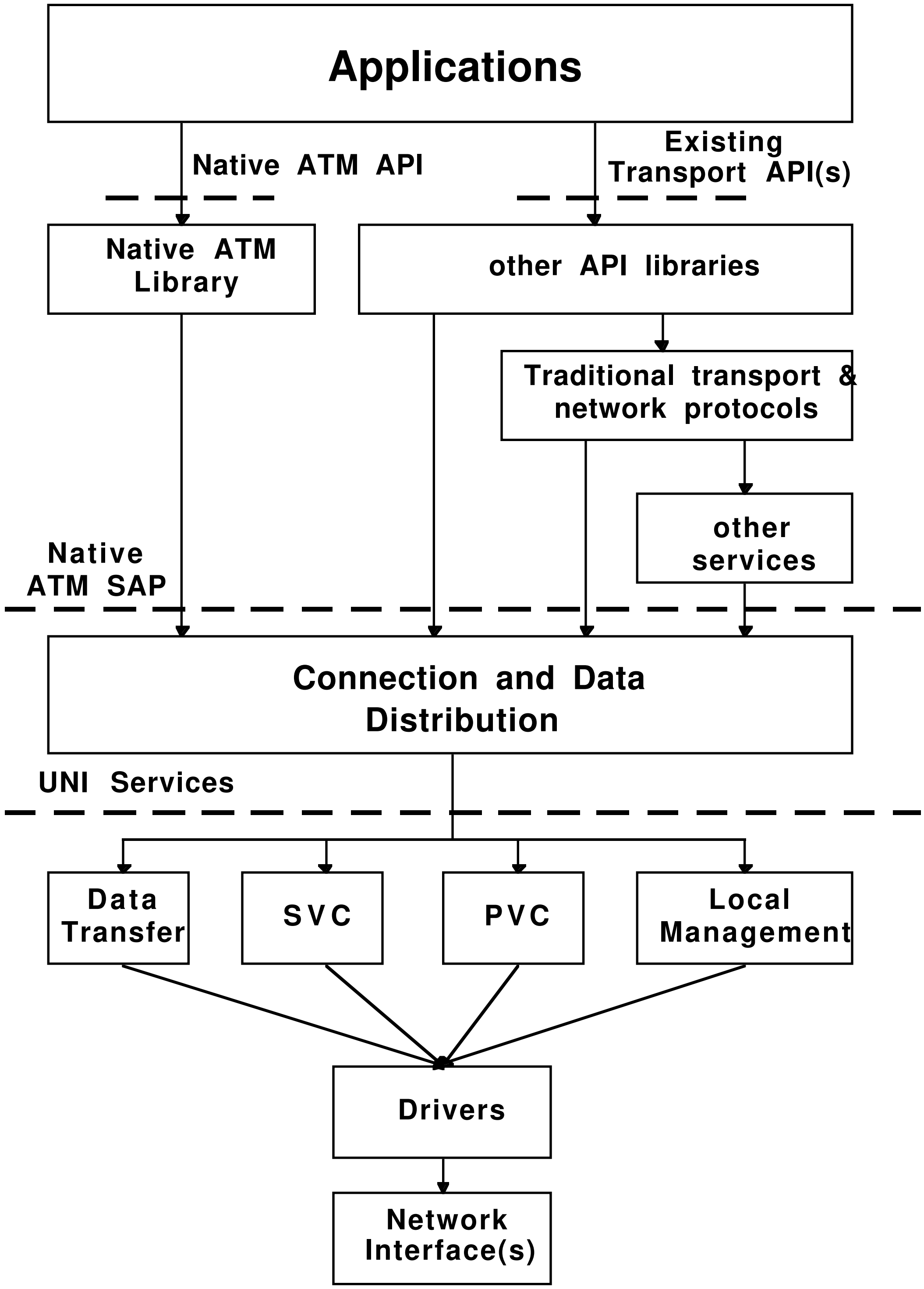


Figure 1: Reference Model for Native ATM Services (Native ATM SAP)

1.3.1 Notes for the Reference Model

The notes in this section are intended to help the reader understand the reference model.

1. “Native ATM API” — an API that is specifically tailored to support Native ATM Services.

2. “Existing Transport APIs” — an API that provides application access to an underlying transport layer. Note that in order to support applications that are aware of an underlying ATM network, the API must be extended to support Native ATM Services. Examples of existing transport APIs include sockets, XTI, Winsock, Netbios, etc.

3. “Native ATM Library” — software component that presents the Native ATM API to an application.

4. “other API libraries” — software components that present the transport APIs to the application.

5. “traditional transport & network protocols” — traditional data communications protocols such as X.25, TCP/IP, SPX/IPX, SNA, Netbui, Appletalk, etc. Note that these protocols may or may not be aware of the underlying ATM network.

6. “other services” — software components that provide an appearance of some network type other than ATM. Examples include ATM LAN Emulation, Circuit Emulation, etc.

7. “Connection and Data Distribution” — this component allows multiple applications (software programs and hardware) to simultaneously use the functions of ATM networks in a native fashion.

8. The direction of the arrows in the diagram are meant to loosely show the relationship between a service provider and a service user. For example, "Applications" uses the services of "Native ATM API".

1.4 Normative Versus Informative Text

The main body of this specification, plus the annexes, are considered normative text. If a particular implementation supports a given service (e.g. SVC, PVC, AAL5, AAL1, SSCOP, ILMI, OAM), then access to that service shall be done in a manner consistent with this specification.

The appendices of this specification are considered informative text.

2. API_connection State Machine

2.1 Motivation

This document describes behavior that is dynamic in nature. The valid actions across the interface to Native ATM Services change with time. For example, an application may not transfer data to another ATM device until an ATM call (between the two ATM devices) is setup across the network. In order to accomplish many actions, a certain sequence of requests and notifications must occur across the interface to Native ATM Services. To be precise, this document describes the dynamic behavior in terms of a finite state machine.

2.2 One-Shot State Machine

Most finite state machines contain cycles, or return paths, because the object being modeled repeats a similar sequence of actions. With these kinds of objects, a “dead end” in the state transition path implies a deadlocked state. The object becomes “stuck” in the deadlocked state and is unavailable for future use. For this reason, the last state in an operational sequence often returns to some initial state.

The state machine used for Native ATM Services is a “one-shot” state machine. After the last state in the operational sequence, the state machine remains in state A11 forever.

The difference in these two approaches result from differences in the objects being modeled. The first case is used when modeling an object that can be reused or recycled. Examples of this are communication channels, blood cells carrying oxygen, or a rental video tape. The second case can be used when modeling an object that has a finite *lifespan*. Examples of this are human age progression or consumption by fire.

By design, the state machine used for Native ATM Services models an object with a finite lifespan. The object being modeled is a connection between two or more parties across an ATM network, which occurs once in time. This approach is superior to the alternative.

The alternative would have been to model a *capacity* provided by the ATM device’s operating system to communicate across the ATM network. In real operating systems, this capacity is typically identified by something analogous to a UNIX file descriptor. The problem with this approach is that various operating systems and existing APIs attach different semantics to the capacity identifier. The differences might include ability to share the capacity identifier across process boundaries, generation of a new capacity identifier when an incoming call is accepted, and the recycling of a capacity identifier after a connection has been terminated. The differing semantics for capacity identifiers would have resulted in a state machine that was tied to a particular operating environment or existing API.

Native ATM Services abstracts away from these differences. Instead, the state machine models an object much more fundamental to ATM communication: an individual connection across the ATM network. The focus of the state machine is an abstract concept herein called an “API_connection” or “connection”. Note that the definition for “connection” differs in this document from the use of that term found in the UNI specification. The precise definition and properties of an API_connection were created in order to describe the interface to Native ATM Services in terms of a finite state machine. The specific definition for an API_connection is found immediately below.

2.3 Terms Used

- “API_endpoint” or “endpoint” - an object associated with a set of attributes by which an application communicates with other ATM devices.

- "API_connection" or "connection" - a relationship between an API_endpoint and other ATM devices, that has the following characteristics:

  1. Data communication may occur between the API_endpoint and the other ATM devices comprising the API_connection.

  2. Each API_connection may occur over a duration of time only once; the same set of communicating ATM devices may form a new connection after a prior connection is released.

  3. The API_connection may be presently active (able to transfer data), or merely anticipated for the future.

2.4 States of API_connection

2.4.1 Null (A0)

There is no association between the API_connection and the API_endpoint that may be used in the future to offer native ATM service to the application.

2.4.2 Initial (A1)

An association now exists between the API_connection and the API_endpoint that may be used to offer native ATM service to the application.

2.4.3 Outgoing Call Preparation (A2)

The application has indicated the intention of placing an outgoing call across the ATM network. Any data structures that are needed to specify characteristics of the outgoing call must exist while the API_connection is in state A2. Those data structures are initialized, to the extent possible, when the API_connection enters state A2. The application is allowed to examine and/or modify some subset of these data structures while in state A2.

2.4.4 Outgoing Call Requested (A3)

The application has requested that an outgoing call be placed across the ATM network. The characteristics of the call are those specified by data structures that were manipulated while the API_connection was in state A2.

State A3 is entered when the application issues a primitive requesting that the outgoing call be established. This state (A3) can be mapped to the following possible system implementations:

polling: The application polls to determine the status of the requested outgoing call. While polling (and before receiving a status indication), the API_connection remains in state A3.

blocking: The operating system blocks the application's process thread, until the outgoing call attempt is either accepted or rejected by the network. While the application is blocked, the API_connection remains in state A3.

messaging: The application receives an asynchronous message to indicate either acceptance or rejection of the outgoing call attempt. While the application is waiting for that message, the API_connection remains in state A3.

2.4.5 Incoming Call Preparation (A4)

The application has indicated the intention of receiving an incoming call across the ATM network. Any data structures that are needed to queue incoming calls (for the purpose of presenting them to the application) must exist while the API_connection is in states A4, A5, A6, and A7.

2.4.6 Wait Incoming Call (A5)

The application has requested that incoming calls from the ATM network be queued for possible acceptance. The characteristics of the queue of potential calls are those specified by data structures that were manipulated while the API_connection was in state A4.

State A5 is entered when the application issues a primitive requesting that incoming calls with the registered SAP be queued and presented to this application. This state (A5) can be mapped to the following possible system implementations:

polling: The application polls to determine the presence of an incoming call that has not been presented to the application. While polling, the API_connection remains in state A5 until the application is notified of the new incoming call.

blocking: The operating system blocks the application's process thread, until a new incoming call is present. While the application is blocked, the API_connection remains in state A5.

messaging: The application receives an asynchronous message to indicate that a new incoming call is present. While the application is waiting for that message, the API_connection remains in state A5.

2.4.7 Incoming Call Present (A6)

An incoming call exists that has not been presented to an application; it is presented to the application for possible acceptance. While in this state (A6), the Native ATM Services makes available characteristics of the incoming call that help the application make an accept/reject decision. If parameter negotiation is present, the application can choose and/or modify the appropriate parameters.

2.4.8 Incoming Call Requested (A7)

The application has accepted the incoming call that is at the head of the incoming call queue. However, the call must be awarded by the network before the call is completely setup and ready for data transfer.

State A7 is entered when the application issues a primitive requesting that the incoming call at the head of the incoming call queue be accepted. This state (A7) can be mapped to the following possible system implementations:

polling: The application polls to determine the status of the accepted call. While polling (and before receiving a status indicating the call was either awarded or released), the API_connection remains in state A7.

blocking: The operating system blocks the application's process thread, until the accepted call is either awarded or released. While the application is blocked, the API_connection remains in state A7.

messaging: The application receives an asynchronous message to indicate that the call is either awarded or released. While the application is waiting for that message, the API_connection remains in state A7.

2.4.9 Point-to-Point Data Transfer (A8)

The point-to-point call (either outgoing or incoming) has been set up across the ATM network. The application may now send and receive data. If any additional states are required for flow control, they will be considered sub-states of state A8.

2.4.10 Point-to-Multipoint Root Data Transfer (A9)

The point-to-multipoint call has been set up across the ATM network, with the application being the root node. The application may now send data. If any additional states are required for flow control, they will be considered sub-states of state A9. If any additional states are required for the joining and releasing of additional leafs, they will be considered sub-states of state A9.

2.4.11 Point-to-Multipoint Leaf Data Transfer (A10)

The point-to-multipoint call has been set up across the ATM network, with the application being a leaf node. The application may now receive data. If any additional states are required for flow control, they will be considered sub-states of state A10.

2.4.12 Connection Terminated (A11)

The connection's lifespan is completed. This state could have been reached by either:

1. An active connection (API_connection in state A8, A9, or A10) was released.
2. A pending connection (API_connection in state A3 or A7) was released by the network or remote ATM device.
3. A connection was aborted by the application.

There is no longer an association between the API_connection and the API_endpoint that was used to offer native ATM service to the application.

2.5 Entity-Relationship Diagrams

The following diagrams are entity-relationship diagrams for each state of API_connection. The prime purpose of the entity-relationship diagrams is to show, for each state:

- the data structures required to implement Native ATM Services
- the information available to the application, through the interface primitives.

In the diagrams, the following conventions are used:

- the rectangles are entities
- the circles are attributes of the attached entities
- the diamonds are relationships between the entities
- the numbers on either side of the diamond denote the cardinality of the relationship (e.g. one-to-one, many-to-one).

2.5.1 Client-Server modeling

Note the cardinality of the relationship between API_endpoint and API_connection. During the early states of the state transition path, the relationship is one-to-many. This means that many API_connections are "bundled" together with one API_endpoint. During the later states of the state transition path, the relationship is one-to-one. Thus, only one API_connection is associated with an API_endpoint.

This is done to model the manner in which client-server programs are written. Typically, a server program will listen for incoming calls on a single API_endpoint. When an incoming call is accepted, then the server program will create a child process that communicates with the client program, but communication will involve a new API_endpoint. Thus, an API_endpoint is created while the API_connection transitions into a data transfer state.

To handle this, the model allows API_connection to change the API_endpoint with which it is associated, during the life of API_connection. The future API_endpoint with which API_connection will be associated during the data transfer phase may be unknown during the early states of the state transition path.

As an example, consider that API_connections labeled X, Y, and Z are going to occur some time in the future, involving a server program listening for incoming calls on API_endpoint 22. The state for API_connections X, Y, and Z would be state A5; and all of them would be associated with API_endpoint 22. As incoming calls are accepted, the following state transitions occur:

1. API_connection X moves to state A8 and becomes associated with API_endpoint 23. Connections Y and Z remain in state A5 and are associated with endpoint 22.

2. API_connection Y moves to state A8 and becomes associated with API_endpoint 24. Connection Z remains in state A5 and is associated with endpoint 22.

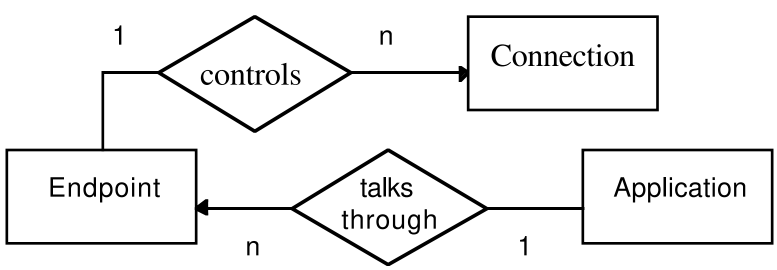
3. API_connection Z moves to state A8 and becomes associated with API_endpoint 25. Any other connections that will be forked from the server program remain in state A5 are associated with endpoint 22.

2.5.2 Specific ER Diagrams for Each State

2.5.2.1 Null (A0)

In this state, there is no association between the API_connection and the API_endpoint that may be used in the future to offer native ATM service to the application. Also, there is no relationship between the application and the API_endpoint. Thus, there is no entity-relationship diagram for this state.

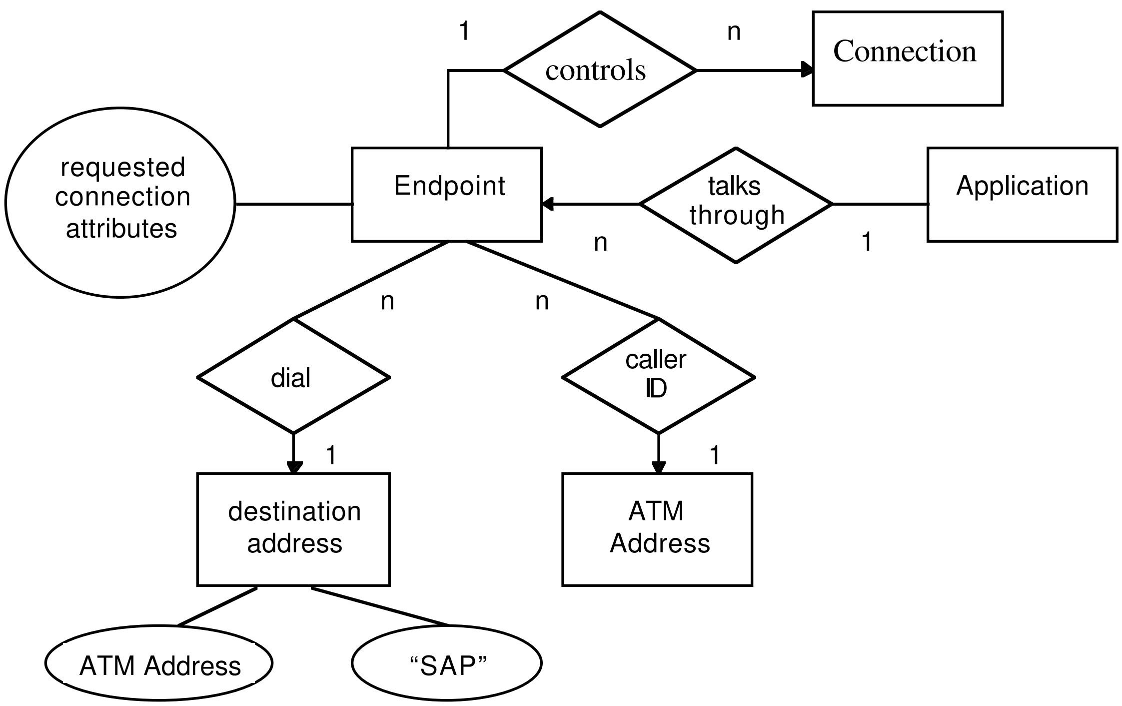
2.5.2.2 Initial (A1)



In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-many. A single API_endpoint may be associated with multiple API_connections, but each API_connection is associated with a maximum of one API_endpoint. The significance of this is that many API_connections may share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

2.5.2.3 Outgoing Call Preparation (A2)



In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-many. A single API_endpoint may be associated with multiple API_connections, but each API_connection is associated with a maximum of one API_endpoint. The significance of this is that many API_connections may share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

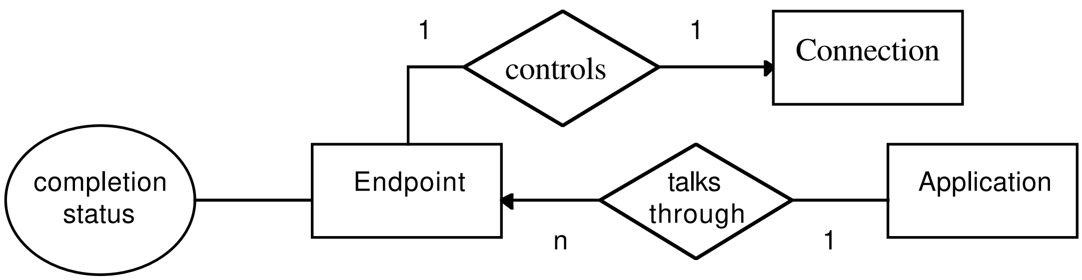
There exist "requested connection attributes" that specify various characteristics of the call to be originated across the ATM network. These are modeled as an attribute of API_endpoint. The application issues primitives to query or set the attributes. Examples of these attributes would include AAL type, forward peak cell rate, quality of service class, and AAL1 clock frequency recovery method.

The API_endpoint is associated with a destination address, which is the complete specification for the software entity with which the application desires to communicate. The destination address includes both the ATM address of the target device, plus a “SAP” that allows the call notification to reach the correct software entity within that target device. See section 4.6 for more information on call distribution within an ATM device. Notice the many-to-one relationship implies that many API_endpoints can simultaneously be communicating with the same destination address.

NOTE: The destination address is supplied by the primitive that move the connection from this state to A3..

In cases where the API_endpoint resides in a device known by multiple ATM addresses, there can only be one ATM address associated with each API_endpoint. The ATM address referenced here will be reported in the “Calling Party Number” signaling information element. This ATM address might be known implicitly by default, or it could be explicitly specified by the application. Notice the many-to-one relationship implies that many API_endpoints can simultaneously be calling from the same ATM address.

2.5.2.4 Outgoing Call Requested (A3)

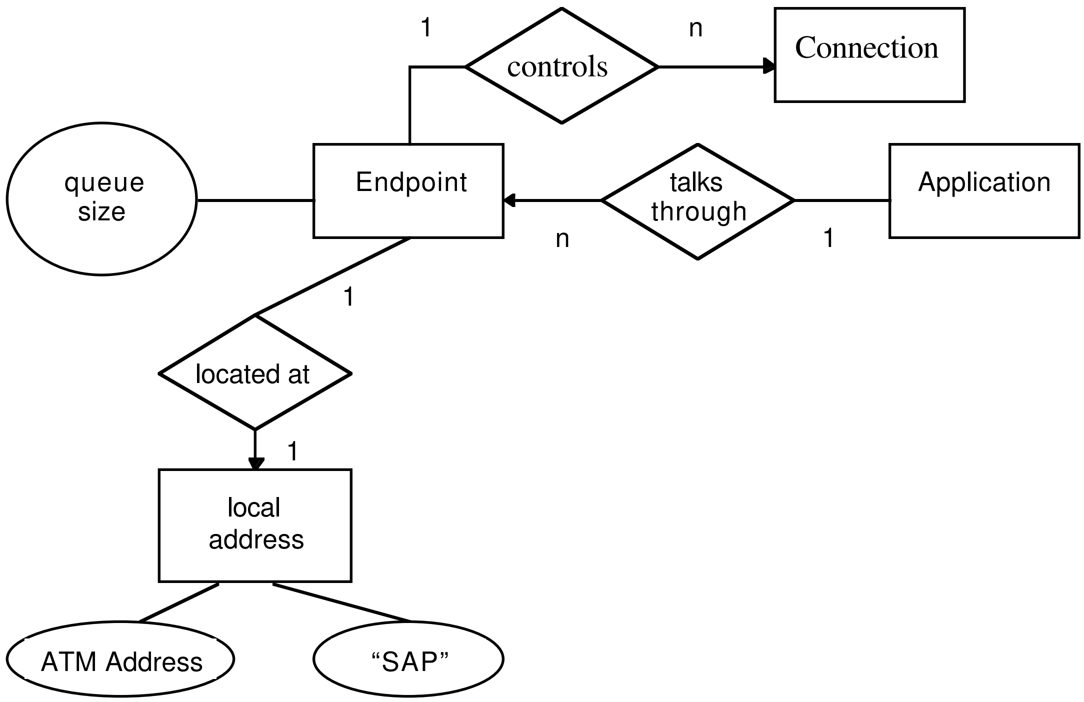


In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-one. Each API_endpoint may be associated with a maximum of one API_connection, and vice-versa. The significance of this is that multiple API_connections may not share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

The attribute of API_endpoint labeled “completion status” displays the status of the application’s request to originate an outgoing call. This modeling may be useful for operating environments where polling is used to implement this state.

2.5.2.5 Incoming Call Preparation (A4)



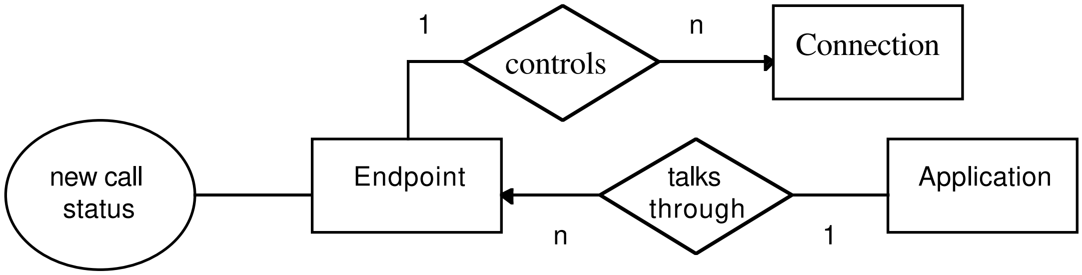
In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-many. A single API_endpoint may be associated with multiple API_connections, but each API_connection is associated with a maximum of one API_endpoint. The significance of this is that many API_connections may share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

The queue size specifies the number of incoming calls that may be held prior to the application's acceptance.

The API_endpoint is associated with a local address, which is the complete incoming call distribution specification for the application. The local address includes both the ATM address of the device housing the application, plus a "SAP" that allows the call notification to reach the correct application within the target device. See section 4.6 for more information on call distribution within an ATM device. Notice the one-to-one relationship ("located at") implies that only one API_endpoint can receive calls on any given local address.

2.5.2.6 Wait Incoming Call (A5)

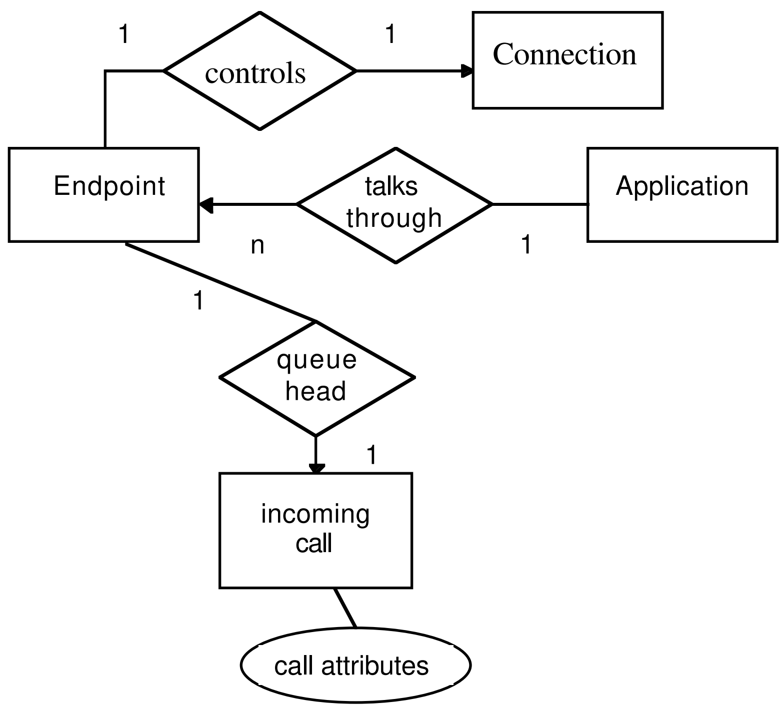


In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-many. A single API_endpoint may be associated with multiple API_connections, but each API_connection is associated with a maximum of one API_endpoint. The significance of this is that many API_connections may share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

The attribute of API_endpoint labeled “new call status” displays the existence of a new incoming call. This modeling may be useful for operating environments where polling is used to implement this state.

2.5.2.7 Incoming Call Present (A6)

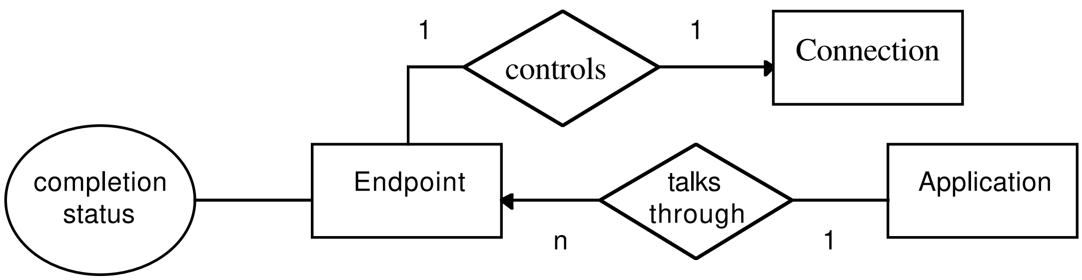


In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-one. Each API_endpoint may be associated with a maximum of one API_connection, and vice-versa. The significance of this is that multiple API_connections may not share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

The queue that holds incoming calls for the application will have one and only one incoming call at the head of the queue. There exist "call attributes" that specify various characteristics of the call being received across the ATM network. These are modeled as an attribute of the entity "incoming call". The application issues primitives to query or set the attributes. Examples of these attributes would include AAL type, forward peak cell rate, quality of service class, and AAL1 clock frequency recovery method. Note that the number of attributes that may be set by the application are the set of parameters that are negotiated end-to-end: "Broadband low-layer information" is an example.

2.5.2.8 Incoming Call Requested (A7)

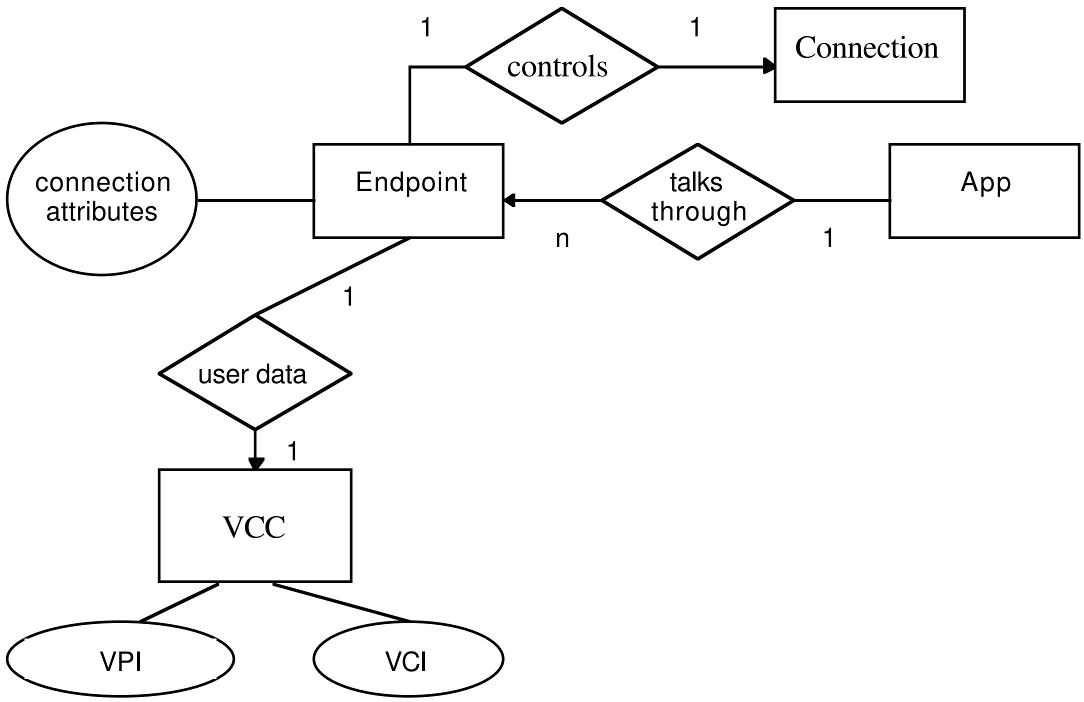


In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-one. Each API_endpoint may be associated with a maximum of one API_connection, and vice-versa. The significance of this is that multiple API_connections may not share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

The attribute of API_endpoint labeled "completion status" displays the status of the application's request to accept an incoming call. This modeling may be useful for operating environments where polling is used to implement this state.

2.5.2.9 Point-to-Point Data Transfer (A8)



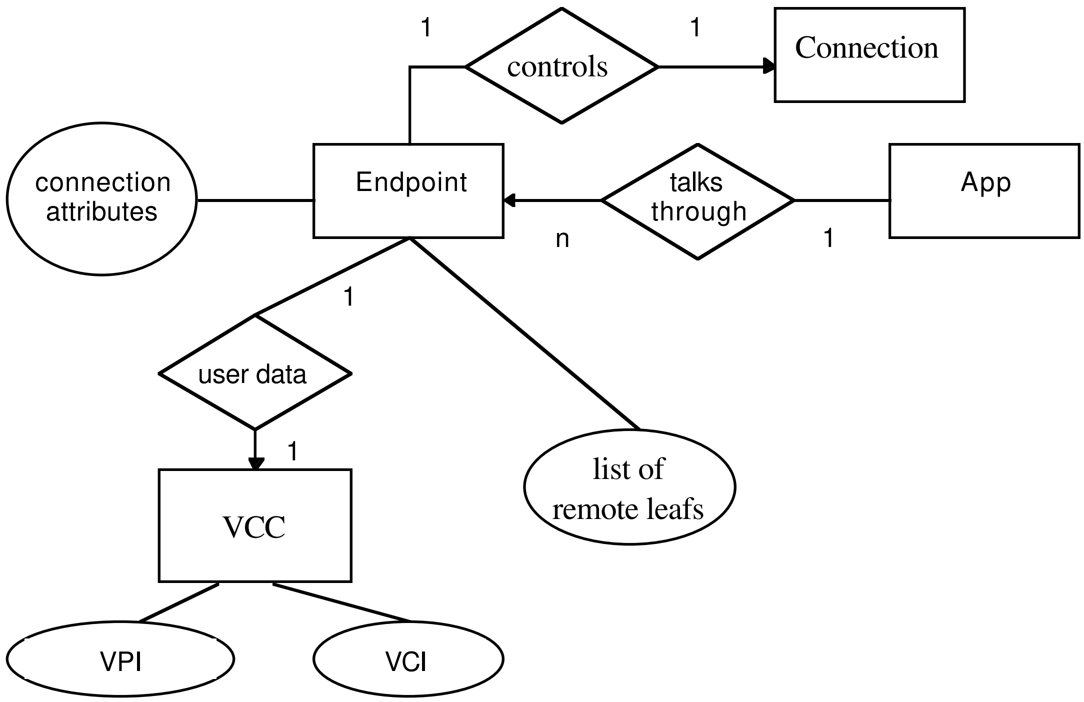
In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-one. Each API_endpoint may be associated with a maximum of one API_connection, and vice-versa. The significance of this is that multiple API_connections may not share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

There exist “connection attributes” that specify various characteristics of the present call. These are modeled as an attribute of API_endpoint. The application issues primitives to query the attributes. Examples of these attributes would include AAL type, forward peak cell rate, quality of service class, and AAL1 clock frequency recovery method.

Every API_endpoint in this state is associated with one and only one VCC that carries user data to/from the application. The VCC may be designated by a VPI and a VCI.

2.5.2.10 Point-to-Multipoint Root Data Transfer (A9)



In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

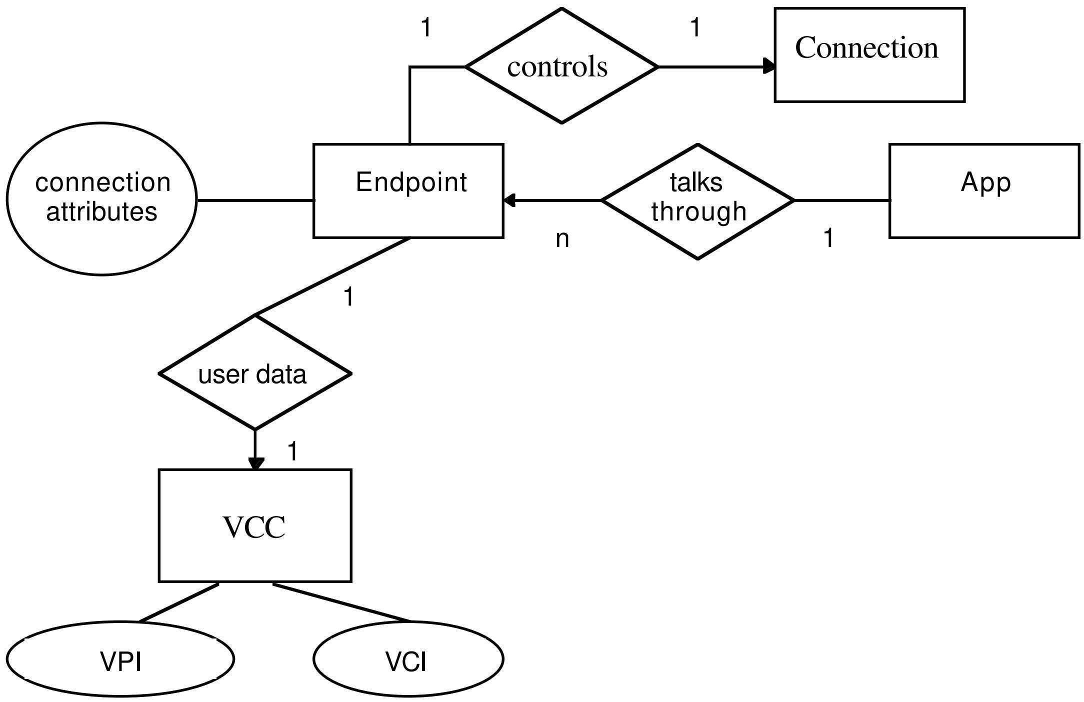
In this state, the relationship between an API_endpoint and an API_connection is one-to-one. Each API_endpoint may be associated with a maximum of one API_connection, and vice-versa. The significance of this is that multiple API_connections may not share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

There exist "connection attributes" that specify various characteristics of the present call. These are modeled as an attribute of API_endpoint. The application issues primitives to query the attributes. Examples of these attributes would include AAL type, forward peak cell rate, quality of service class, and AAL1 clock frequency recovery method.

Every API_endpoint in this state is associated with one and only one VCC that carries user data to/from the application. The VCC may be designated by a VPI and a VCI.

A "list of remote leafs" may be modeled as an attribute of API_endpoint. The remote leafs are other ATM devices that are leafs of a point-to-multipoint call, with the device housing API_endpoint as the root. The application uses primitives to modify this list, thereby adding and dropping parties of the point-to-multipoint call.

2.5.2.11 Point-to-Multipoint Leaf Data Transfer (A10)



In this state, the application has access to an API_endpoint. Native ATM Services are offered to the application via the exchange of primitives between the application and the API_endpoint. The relationship between an application and the API_endpoint is one-to-many; an application may exchange primitives with many API_endpoints, but each API_endpoint is bound to a maximum of one application.

In this state, the relationship between an API_endpoint and an API_connection is one-to-one. Each API_endpoint may be associated with a maximum of one API_connection, and vice-versa. The significance of this is that multiple API_connections may not share a single API_endpoint in this state. This is done to model environments where an API_connection acquires a new API_endpoint as the API_connection progresses from the early states (call setup) to the data transfer states (A8, A9, A10).

There exist "connection attributes" that specify various characteristics of the present call. These are modeled as an attribute of API_endpoint. The application issues primitives to query the attributes. Examples of these attributes would include AAL type, forward peak cell rate, quality of service class, and AAL1 clock frequency recovery method.

Every API_endpoint in this state is associated with one and only one VCC that carries user data to/from the application. The VCC may be designated by a VPI and a VCI.

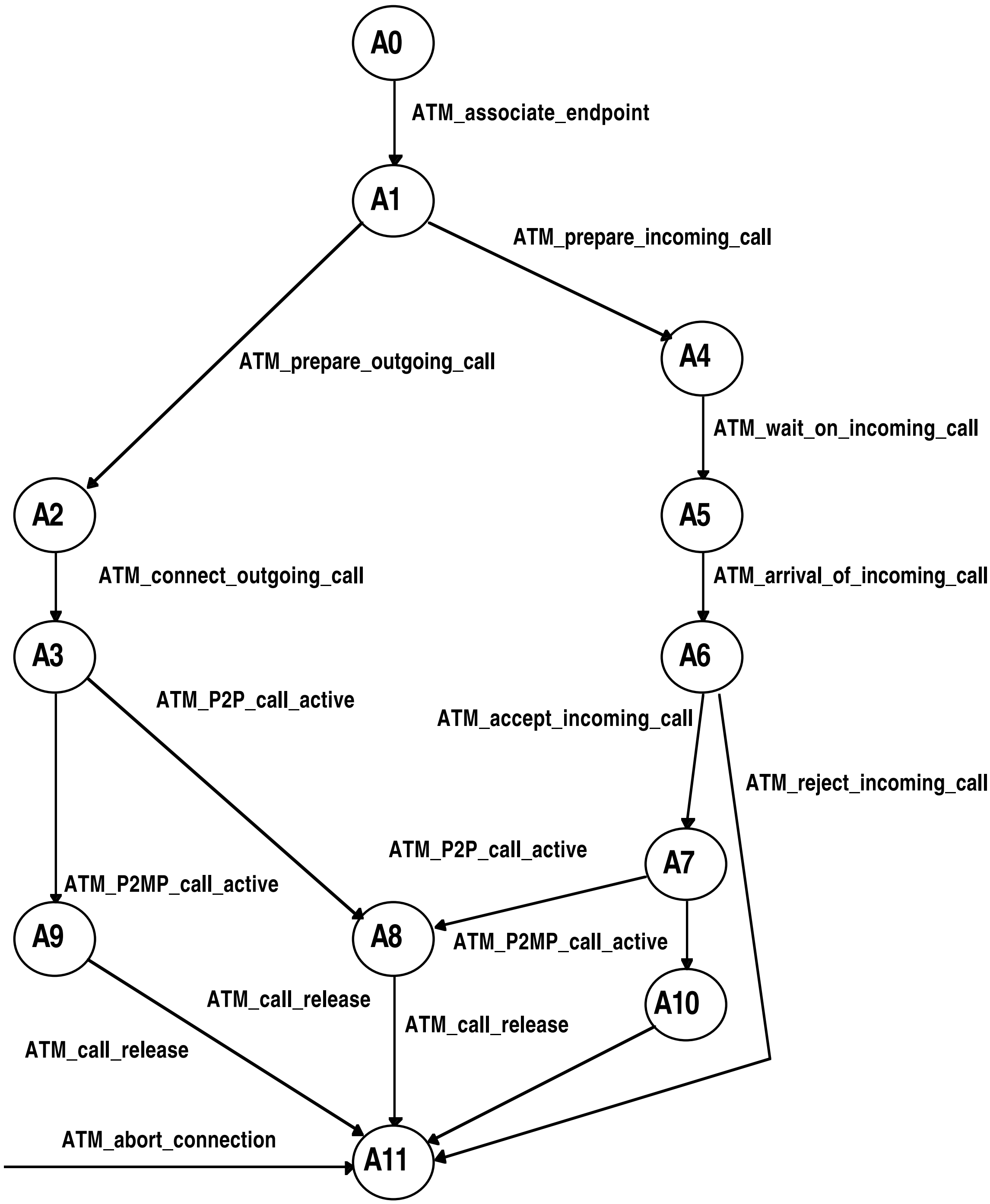
2.5.2.12 Connection Terminated (A11)

There is no entity-relationship diagram for this state.

The lifetime of API_connection is now over. There is no remaining association between an API_endpoint and the API_connection. Hence, there is no entity-relationship diagram needed for this state.

2.6 Partial State Diagram

Below is a partial state diagram for API_connection. The circles are the states and the edges are primitives that cause transitions between states. These primitives will be described in chapter 3.



3. Primitives

This section uses the following conventions for the primitives specified herein:

- "IN" - a parameter value supplied by the originator of the primitive (e.g. the caller of a subroutine).
- "OUT" - a parameter value supplied by the recipient of the primitive (e.g. the callee of a subroutine).
- "INOUT" - a parameter value that might be supplied by the primitive's originator, recipient, or both; depending on the implementation.
- "Request", "Indication", "Response", "Confirm" - these terms are conventionally used by ISO to describe the role of the primitive.

3.1 Control Plane

ATM_abort_connection **Request**

Purpose: clear the connection due to abnormal conditions

ATM_abort_connection (
IN *endpoint_identifier*,
IN *cause*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *cause* specifies the abnormal condition

Return Values: none

Valid States: A1, A2, A3, A4, A5, A6, A7, A8, A9, A10

State Transitions: the state becomes A11

This primitive ends the association between the API_connection and an API_endpoint.

This primitive is used to terminate a communication endpoint. All connections associated with this communication endpoint will be aborted and all the resources allocated for the communication endpoint are released.

ATM_accept_incoming_call **Response**

Purpose: application signals acceptance of the pending incoming call

ATM_accept_incoming_call (
IN *endpoint_identifier*,
INOUT *new_endpoint_identifier*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *new_endpoint_identifier* specifies the new endpoint identifier to be associated with this API_connection. In cases where the parent application provides a new endpoint identifier, this parameter is IN. In case where the underlying service provides a new endpoint identifier, this parameter is OUT.

Return Values:

SUCCESS

CONNECTION_PREVIOUSLY_ABORTED

Valid States: A6

State Transitions:

- If the return value is SUCCESS, then the API_connection most recently presented to the application moves to state A7.
- If the return value is CONNECTION_PREVIOUSLY_ABORTED, then the API_connection most recently presented to the application moves to state A11.

This primitive signals that the application wishes to accept the incoming call that is at the head of the incoming call queue.

The connection attribute BLLI_SELECTOR is set by the application to one of the following values:

- 0: no “Broadband Low Layer Information” (BLLI) information element should be returned to the calling party
- 1: the first occurrence of BLLI information element should be returned to the calling party
- 2: the second occurrence of BLLI information element should be returned to the calling party
- 3 - the third occurrence of BLLI information element should be returned to the calling party.

ATM_add_party **Request**

Purpose: add a leaf to an existing point-to-multipoint call

ATM_add_party (

IN *endpoint_identifier***,**

IN *leaf_identifier***,**

IN *leaf_ATM_address*

)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *leaf_identifier* is reference to the ATM device being added to this connection. (NOTE: This value must be non-zero and less than 32,768.)
- *leaf_ATM_address* is the ATM address of the ATM device being added to this connection.

Return Values: none

Valid States: A9

State Transitions: none

This primitive allows applications to add new ATM devices to an existing point-to-multipoint call. The new leaf inherits the traffic parameters already in force for the existing leaf(s) of this connection. Note that data traffic in a point-to-multipoint connection only flows from the root node to each leaf node.

ATM_add_party_reject **Confirm**

Purpose: signal that a previous ATM_add_party request was unsuccessful.

ATM_add_party_reject (

IN *endpoint_identifier***,**

IN *leaf_identifier***,**

IN *rejection_cause*

)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *leaf_identifier* is a reference to the ATM device that could not be added.
- *rejection_cause* specifies why the addition of a leaf was rejected.

Return Values: none

Valid States: A9

State Transitions: no transitions

ATM_add_party_success **Confirm**

Purpose: signal that a previous ATM_add_party request was successful.

ATM_add_party_success (
IN *endpoint_identifier*,
IN *leaf_identifier*,
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *leaf_identifier* is reference to the ATM device being added to this connection.

Return Values: none

Valid States: A9

State Transitions: no transitions

ATM_arrival_of_incoming_call **Indication**

Purpose: notifies the application that a call has arrived

ATM_arrival_of_incoming_call (
IN *endpoint_identifier*,
)

where

- *endpoint_identifier* specifies the incoming call queue upon which the call is located.

Return Values: none

Valid States: A5

State Transitions: The API_connection that is presented to the application moves from state A5 to A6. All other API_connections for this endpoint_identifier remain in state A5.

This primitive signals the application that an incoming call is present in the incoming call queue.

The connection attribute BLLI_SELECTOR is set by Native ATM Services to one of the following values:

- 0: no "Broadband Low Layer Information" (BLLI) information element was present in the incoming call
- 1: the first occurrence of BLLI information element present in the incoming call forms part of the SAP (over which this call is being received)

- 2: the second occurrence of BLLI information element present in the incoming call forms part of the SAP (over which this call is being received)
- 3 - the third occurrence of BLLI information element present in the incoming call forms part of the SAP (over which this call is being received).

ATM_associate_endpoint **Request**

Purpose: associates an API_endpoint with the potential API_connections that will use that endpoint.

ATM_associate_endpoint (
  OUT *endpoint_identifier***,**
)

where

- *endpoint_identifier* is the newly allocated API_endpoint.

Return Values: none

Valid States: A0

State Transitions: API_connection moves to state A1

This primitive creates an initial association between the API_connection and API_endpoint.

ATM_call_release **Request**

Purpose: terminates an API_connection that is in an active state (A8, A9, A10) by the application.

ATM_call_release (
  IN *endpoint_identifier***,**
  IN *release_cause***,**
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *release_cause* identifies the cause of release.

Return Values: none

Valid States: A8, A9, A10

State Transitions: API_connection moves to state A11

ATM_call_release **Indication**

Purpose: terminates an API_connection because of a network or remote device action.

ATM_call_release (
IN *endpoint_identifier*,
IN *release_cause*,
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *release_cause* identifies the cause of release.

Return Values: none

Valid States: A3, A7, A8, A9, A10

State Transitions: API_connection moves to state A11

ATM_connect_outgoing_call **Request**

Purpose: initiates a call across the ATM network

ATM_connect_outgoing_call (
IN *endpoint_identifier*,
IN *destination_SAP,*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *destination_SAP* is the address of the target ATM device and target entity within the ATM device. See section 4.4 for more information on SAP addresses.

Return Values: none

Valid States: A2

State Transitions: API_connection moves to state A3

This primitive causes the underlying ATM protocols to attempt set up of an outgoing call.

ATM_drop_party — Request

Purpose: remove an ATM device from a point-to-multipoint call.

ATM_drop_party (
IN *endpoint_identifier*,
IN *leaf_identifier*,
IN *drop_cause*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *leaf_identifier* specifies the ATM device being dropped.
- *drop_cause* specifies the reason for dropping.

Return Values: none

Valid States: A9

State Transitions: no state transitions

This primitive allows applications to drop a leaf node from a point-to-multipoint connection.

ATM_drop_party — Indication

Purpose: an ATM device (leaf of point-to-multipoint call) released the connection.

ATM_drop_party (
IN *endpoint_identifier*,
IN *leaf_identifier*,
IN *drop_cause*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *leaf_identifier* specifies the ATM device that dropped off of the connection.
- *drop_cause* specifies the reason for dropping.

Return Values: none

Valid States: A9

State Transitions: no state transitions

This primitive notifies an application that a remote leaf node dropped off a point-to-multipoint connection.

ATM_get_local_port_info **Request**

Purpose: obtain necessary information related to an ATM physical port

ATM_get_local_port_info (
IN *port_number*,
OUT *address_list*
OUT *line_rate*
)

where

- *port_number* identifies a physical UNI attachment.
- *address_list* is a list of ATM addresses that are valid for the port number.
- *line_rate* is the maximum cell rate supported by the ATM port in cells per second.

Return Values:
SUCCESS
INVALID_PORT_NUMBER
NO_VALID_ADDRESSES

Valid States: not applicable

State Transitions: not applicable

This primitive reports the result of ILMI address registration, as specified in section 5.8 of the UNI.

ATM_P2MP_call_active **Confirm**

Purpose: signal that the point-to-multipoint call is now in active state

ATM_P2MP_call_active (
IN *endpoint_identifier*,
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.

Return Values: none

Valid States: A3, A7

State Transitions: If the API_connection was in state A3, then the new state will be A9. If the API_connection was in state A7, then the new state will be A10.

ATM_P2P_call_active — Confirm

Purpose: signal that the point-to-point call is now in active state

ATM_P2P_call_active (
  IN *endpoint_identifier*,
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.

Return Values: none

Valid States: A3, A7

State Transitions: API_connection moves A8.

ATM_prepare_incoming_call — Request

Purpose: sets up incoming call distribution tables

ATM_prepare_incoming_call (
  IN *endpoint_identifier,*
  IN *local_SAP,*
  IN *queue_size*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *local_SAP* is the address of this ATM device and application entity. See section 4.4 for a discussion on SAP addresses.
- *queue_size* specifies the depth of the incoming call queue.

Return Values:
SUCCESS
SAP_ALREADY_USED

Valid States: A1

State Transitions: API_connection moves to state A4

This primitive allows applications to associate an API_endpoint with a local address. The intent of the 2nd parameter is to specify an address that can be used to identify a unique API_endpoint.

ATM_prepare_outgoing_call — Request

Purpose: sets up data structures that hold the characteristics of the outgoing call

ATM_prepare_outgoing_call (
IN *endpoint_identifier,*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.

Return Values: none

Valid States: A1

State Transitions: API_connection moves to state A2.

ATM_query_connection_attributes — Request

Purpose: to obtain an attribute value of the connection

ATM_query_connection_attributes (
IN *endpoint_identifier,*
IN *attribute_name,*
OUT *attribute_value,*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *attribute_name* identifies the attribute.
- *attribute_value* is the value of attribute.

Return Values:
SUCCESS
ATTRIBUTE_DOES_NOT_EXIST

Valid States: A2, A6, A8, A9, A10

State Transitions: no state transitions

This primitive allows applications to query connection attributes. The attribute_name parameter specifies the name of the attribute (e.g. QoS, peak cell rate). The attribute_value parameter contains the query result.

NOTE: Native ATM Services must provide access to as many as three instances of the connection attributes found in the "Broadband Low Layer Information" information element. One convenient way to do this is to use connection attribute BLLI_SELECTOR to select which of the three instances is being queried. Such implementations should default BLLI_SELECTOR to a value of 1 when API_connection transitions from state A1 to A2.

ATM_reject_incoming_call — Response

Purpose: signals that the application does not accept the incoming call

ATM_reject_incoming_call (
IN *endpoint_identifier,*
IN *rejection_cause*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *rejection_cause* specifies why the connection is being rejected.

Return Values: none

Valid States: A6

State Transitions: API_connection moves to state A11

This primitive signals that the application wishes to reject the incoming call that is at the head of the incoming call queue.

ATM_set_connection_attributes — Request

Purpose: to modify an attribute value of the connection

ATM_set_connection_attributes (

IN *endpoint_identifier,*

IN *attribute_name,*

IN *attribute_value,*

)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *attribute_name* identifies the attribute.
- *attribute_value* is the desired value of attribute.

Return Values:
SUCCESS
ATTRIBUTE_DOES_NOT_EXIST
CAN_NOT_MODIFY
INVALID_VALUE

Valid States: A2, A6

State Transitions: no state transitions

This primitive allows applications to modify connection attributes. The attribute_name parameter specifies the name of the attribute (e.g. QoS, peak cell rate). The attribute_value parameter contains the desired new value.

NOTE: Native ATM Services must provide access to as many as three instances of the connection attributes found in the “Broadband Low Layer Information” information element. One convenient way to do this is to use connection attribute BLLI_SELECTOR to select which of the three instances is being manipulated. Such implementations should default BLLI_SELECTOR to a value of 1 when API_connection transitions from state A1 to A2.

ATM_wait_on_incoming_call — Request

Purpose: activates the incoming call distribution function for this endpoint.

ATM_wait_on_incoming_call (

IN *endpoint_identifier,*

)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.

Return Values: none

Valid States: A4

State Transitions: All API_connections associated with this endpoint move to state A5

This primitive causes the application to wait for an incoming call to enter the incoming call queue.

3.2 Data Plane

There are three major approaches to the implementation of the data transfer primitives:

- polling
- blocking
- messaging

Another issue is the location of the data. The data bound for the application could exist in:

- memory space that the operating system kernel manages
- memory space that the application manages
- hardware (e.g. the ATM adapter card, the video card, the sound card, a private data bus)

It is possible that there is a total hardware path between the ATM function and an application implemented in hardware. In this case, the data plane primitives are realized in hardware.

Implementation choices are outside the scope of this document. However, for completeness, this section considers the differing semantics associated with these different implementation approaches. See Appendix A for a discussion of pragmatic issues concerning these primitives.

NOTE: the data plane primitives to support AAL1 and User defined AAL traffic are implementation specific at this time. This topic is for further study.

3.2.1 Sending Data

ATM_send_data **Request**

Purpose: to send data on the API_connection

ATM_send_data (
IN *endpoint_identifier,*
IN *data_source,*
OUT *sending_result*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *data_source* describes the data. For example, it could be a buffer location and amount of data, or even a collection of buffers. As another example, it could be an instance of the application's data to be transmitted across the ATM network.
- *sending_result* is a status indication. In some implementations, it is simply SUCCESS or FAILURE. In other implementations, it could be the amount of data transferred.

Return Values:
SUCCESS
NO_CONNECTION

Valid States: A8, A9

State Transitions: no state transition

3.2.2 Receiving Data

Only one of the following implementations needs to be supported in a conforming system.

3.2.2.1 Polling Implementation

ATM_receive_data **Request**

Purpose: to receive data on the API_connection

ATM_receive_data (
IN *endpoint_identifier,*
INOUT *data_receptor,*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *data_receptor:*

  Before this primitive is invoked, *data_receptor* describes where the received data should be placed. After this primitive returns, *data_receptor* describes the data.

  For example, before the primitive is invoked, *data_receptor* could be a buffer location and buffer size. After the primitive returns, *data_receptor* would be a buffer location and amount of data.

  As another example, before the primitive is invoked, *data_receptor* could be a character string of zero length and a maximum string size. After the primitive returns, data_receptor would be a character string.

Return Values:
SUCCESS
DATA_NOT_PRESENT
NO_CONNECTION

Valid States: A8, A9, A10

State Transitions: no state transition

3.2.2.2 Blocking Implementation

ATM_receive_data **Request**

Purpose: to receive data on the API_connection

ATM_receive_data (
IN *endpoint_identifier,*
INOUT *data_receptor,*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *data_receptor:*

  Before this primitive is invoked, *data_receptor* describes where the received data should be placed. After this primitive returns, *data_receptor* describes the data.

  For example, before the primitive is invoked, *data_receptor* could be a buffer location and buffer size. After the primitive returns, *data_receptor* would be a buffer location and amount of data.

  As another example, before the primitive is invoked, *data_receptor* could be a character string of zero length and a maximum string size. After the primitive returns, *data_receptor* would be a character string.

Return Values:
SUCCESS
NO_CONNECTION

Valid States: A8, A9, A10

State Transitions: no state transition

3.2.2.3 Messaging Implementation

ATM_receive_data **Indication**

Purpose: to receive data on the API_connection

ATM_receive_data (
IN *endpoint_identifier,*
IN *receive_data*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *receive_data* describes the data. Consider the following examples for this parameter:
    1. a buffer location and the amount of data
    2. an instance of the data (received across the ATM network) traversing a bus from the ATM hardware to some other hardware device
    3. an indication to an implementation-specific "receive handler" (see below) that the data either has been or is in the process of being received across the ATM network.

Return Values: none

Valid States: A8, A9, A10

State Transitions: no state transition

In some cases, a "receive handler" would be installed in the system. An environment-specific function is performed to install the receive handler. The receive handler becomes the recipient of the ATM_receive_data primitive. Additionally, the sending application could include application-defined control information in an AAL frame and the application-defined receive handler could interpret this control information.

3.3 Management Plane

ATM_confirm_loopback **Confirm**

Purpose: confirms completion of loopback test

ATM_confirm_loopback (
IN *endpoint_identifier*,
IN *correlator*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *correlator* allows the management application to match this confirmation to a previous ATM_initiate_loopback request.

Return Values: none

When this confirmation is received, it means that the loopback test was successful. If no confirmation to a corresponding ATM_intiate_loopback is received, then the loopback test failed.

ATM_indicate_error **Indication**

Purpose: indicates an error was detected among the managed objects.

ATM_indicate_error (
IN *error_code,*
IN *endpoint_identifier,*
)

where

- *error_code* is the error that occurred.
- *endpoint_identifier* specifies the connection to which this primitive applies, when *error_code* is identifies an error that is associated with a single connection. Otherwise, the value of *endpoint_identifier* is unspecified.

Return Values: none

Error_Code Values:

Note that these are network-oriented error messages. They arise from errors detected over the UNI.

NOTE: these values are for further study

ATM_indicate_fault_alert **Indication**

Purpose: indicates reception of an OAM alarm signal

ATM_indicate_fault_alert (
IN *endpoint_identifier,*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.

Return Values: none

ATM_initiate_loopback **request**

Purpose: initiates a loopback test to the nearest switch or the far end point of a connection.

ATM_initiate_loopback (
IN *endpoint_identifier,*
IN *loopback_extent,*
IN *correlator,*
)

where

- *endpoint_identifier* specifies the connection to which this primitive applies.
- *loopback_extent* specifies nearest switch loopback or end-to-end loopback.
- *correlator* is a value that will be returned with the ATM_confirm_loopback primitive.

Return Values: none

ATM_query_mgmt_variable **Request**

Purpose: queries the value of the specified management variable.

ATM_query_mgmt_variable (

IN *variable_name,*

IN *table_index,*

OUT *variable_value,*

)

where

- *variable_name* identifies the managed object.
- *table_index* identifies the table location when this variable is located in a table. For variables not located in a table, this parameter is ignored.
- *variable_value* is the value of the managed object.

Return Values:
SUCCESS
VARIABLE_NOT_SUPPORTED
INDEX_OUT_OF_RANGE
VARIABLE_DOES_NOT_EXIST

ATM_set_mgmt_variable **Request**

Purpose: modifies the value of the specified management variable

ATM_set_mgmt_variable (

IN *variable_name,*

IN *table_index,*

IN *desired_value,*

)

where

- *variable_name* identifies the managed object.
- *table_index* identifies the table location when this variable is located in a table. For variables not located in a table, this parameter is ignored.
- *desired_value* is the desired value of the managed object.

Return Values:
SUCCESS
VARIABLE_NOT_SUPPORTED
INDEX_OUT_OF_RANGE
VARIABLE_DOES_NOT_EXIST
CAN_NOT_MODIFY
INVALID_VALUE

4. Procedures

4.1 PVC Provisioning

These procedure shall be limited to trusted applications, such as a management application. General user applications that use a PVC shall only receive an indication that a PVC is active.

4.1.1 Establishment

The management application shall perform the following steps in order to provision a PVC:

1. Choose an unused table entry in the atmfVccGroup.

2. Using the **ATM_set_mgmt_variable** primitive, set all the parameters for this table entry of the atmfVccGroup except for atmfVccOperStatus. It is presumed that atmfVccOperStatus has the value unknown or localDown.

3. Using the **ATM_set_mgmt_variable** primitive, set atmfVccOperStatus in atmfVccGroup to localUpEnd2endUnknown. This is a signal to native ATM services to perform all locally required setup of the VC. This includes hardware register manipulation and buffer allocation. During this time atmfVccEndpointIdentifier shall be assigned by native ATM services. The value for atmfVccOperStatus will retain the value of unknown or localDown until the initialization of the VC is complete. After the initialization is completed atmfVccOperStatus is set to localUpEnd2endUnknown by Native ATM services.

4. Using the **ATM_query_mgmt_variable** primitive, poll and query atmfVccOperStatus in atmfVccGroup until the value is localUpEnd2endUnknown.

5. Using the **ATM_query_mgmt_variable** primitive, obtain the value for atmfVccEndpointIdentifier and pass it to the entity that is to use this PVC.

4.1.2 Termination

The management application shall perform the following steps in order to terminate a PVC:

1. Using the **ATM_set_mgmt_variable** primitive, set atmfVccOperStatus in atmfVccGroup to localDown. This is a signal to native ATM services to perform all locally required teardown of the VC. This includes hardware register manipulation and buffer deallocation.

4.2 SVC Provisioning

4.2.1 Initiating a Call

4.2.1.1 Establishment

The application first acquires a local connection endpoint with which primitives may be exchanged that offer native ATM services. This is done via the **ATM_associate_endpoint** primitive. The *endpoint_identifier* that is returned from this primitive is used in the future to identify the newly acquired connection endpoint.

Next, the application must signal its intent to initiate an outgoing call. The application does this via the **ATM_prepare_outgoing_call** primitive. In response to the primitive, the connection endpoint creates data structures that initially hold default values for various connection attributes. Examples of connection attributes include the AAL type, forward peak cell rate, and QOS class; for a complete listing see Annex A. The application may examine any of these default connection attributes via the **ATM_query_connection_attributes** primitive. In addition, the application may optionally modify some of these attributes via the **ATM_set_connection_attributes** primitive. Depending on the implementation, each of these attributes may or may not be settable by the application. Also, the value to which the application attempts to set a given connection attribute may be modified by the connection endpoint, due to local resource constraints or the local implementation of Native ATM Services.

The application can optionally select additional data plane services. The default is that no additional data plane service is provided beyond AAL5. The only such additional option supported at this time is the SSCOP protocol; SSCOP adds reliability to the ATM connection. Option selection is performed via the SSCS connection attribute.

If the application wishes to place a point-to-multipoint call, then the application signals this to the connection endpoint by setting the CONNECT_CONFIG connection attribute to reflect this. The default for CONNECT_CONFIG shall be to select a point-to-point call. When a point-to-multipoint connection is selected, then Native ATM Services will enforce a reverse bandwidth of zero and indicate a leaf identifier of zero in the Q.2931 network messages.

When the application is satisfied that the connection attributes, as represented by the connection endpoint, conform to the application's requirements, then a call is placed across the ATM network. The application initiates this via the **ATM_connect_outgoing_call** primitive. Included as a parameter of this primitive is *destination_SAP*. The destination SAP is the ATM address of the remote ATM device plus the additional information allowing the call to reach the correct target software entity within the remote ATM device.

Native ATM Services places the call across the ATM network to the target ATM device. If a point-to-point call attempt is successful, then Native ATM Services signals the application via an **ATM_P2P_call_active** primitive. If a point-to-multipoint call attempt is successful, then Native ATM Services signals the application via an **ATM_P2MP_call_active** primitive. In the event that the call attempt was unsuccessful, then Native ATM Services signals the application via an **ATM_call_release** primitive. This primitive includes the cause for the lack of success.

4.2.1.2 Adding and Removing Leafs

For point-to-multipoint calls, the application originating the call has the ability to add and remove additional parties of the call. In this section, the originating application is referred to as the "root" of the call; all other participants of the call are referred to as "leafs".

To add a leaf, the root issues an **ATM_add_party** primitive. The parameters for this primitive include the ATM address of the leaf, plus a *leaf_identifier*, which is used to identify the leaf in future primitives that affect that leaf. Note that the application is responsible for generating values of *leaf_identifier*. If the leaf is successfully added to the point-to-multipoint call, then the application is notified of this via the **ATM_add_party_success** primitive. Otherwise, the application is notified that the request to add a leaf failed, via the **ATM_add_party_reject** primitive.

To remove a leaf, the root issues an **ATM_drop_party (request)** primitive. Included as a parameter of this primitive the root's reason for removing the leaf.

If a leaf decides to remove itself from a point-to-multipoint call, then Native ATM Services notifies the application via the **ATM_drop_party (indication)** primitive. Included as a parameter of this primitive is the leaf's reason for removing itself. This primitive would also be used in the case of the ATM network deciding to remove the leaf.

4.2.1.3 Termination

When the application wishes to terminate the connection, then the application invokes the **ATM_call_release (request)** primitive. If the remote ATM device or the ATM network terminates the connection, then the application is notified of this via the **ATM_call_release (indication)** primitive.

4.2.2 Responding to a Call

4.2.2.1 Establishment

The application first acquires a local connection endpoint with which primitives may be exchanged that offer Native ATM Services. This is done via the **ATM_associate_endpoint** primitive. The *endpoint_identifier* that is returned from this primitive is used in the future to identify the newly acquired connection endpoint.

Next, the application must signal its intent to respond to an incoming call. The application does this via the **ATM_prepare_incoming_call** primitive. Included as parameters of this primitive are the ATM address of the local ATM device, plus the SAP information allowing call notification to reach the application. During the processing of this primitive, Native ATM Services ensures that no other application is using the same SAP.

The application then issues the **ATM_wait_on_incoming_call** primitive to request that incoming calls be queued and presented to the application. When such an incoming call does arrive, the connection endpoint notifies the application via the **ATM_arrival_of_incoming_call** primitive.

The application must make a decision as to whether or not to accept the call. The application may examine the connection attributes of the newly arrived incoming call via the **ATM_query_connection_attributes** primitive. In addition, the application may modify a small number of attributes via the **ATM_set_connection_attributes** primitive.

If the application decides to accept the call, the application invokes the **ATM_accept_incoming_call** primitive. Otherwise, the application rejects the call via the **ATM_reject_incoming_call** primitive.

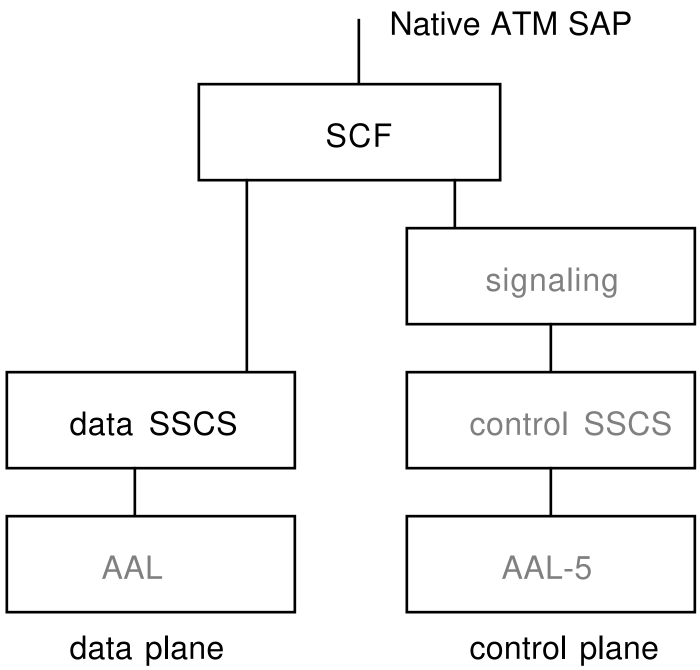
Even if call has been accepted, the application must wait for the ATM network to award the call. After an incoming point-to-point call has been awarded, the application is notified via the **ATM_P2P_call_active** primitive. After an incoming point-to-multipoint call has been awarded, the application is notified via the **ATM_P2MP_call_active** primitive. If an error occurred during the awarding of the call, then the application is notified via the **ATM_call_release** primitive.

4.2.2.2 Termination

When the application wishes to terminate the connection, then the application invokes the **ATM_call_release (request)** primitive. If the remote ATM device or the ATM network terminates the connection, then the application is notified of this via the **ATM_call_release (indication)** primitive.

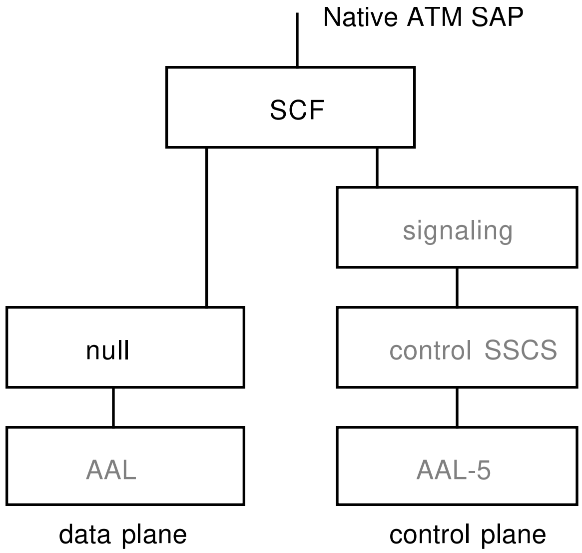
4.3 Synchronization and Coordination Function

This section describes the synchronization and coordination function (SCF) that Native ATM Services performs between the control and data planes. This service is related to and can be selected via the local service connection attribute SSCS that is modified with the *ATM_set_connection_attribute* primitive.



4.3.1 NULL_SSCS

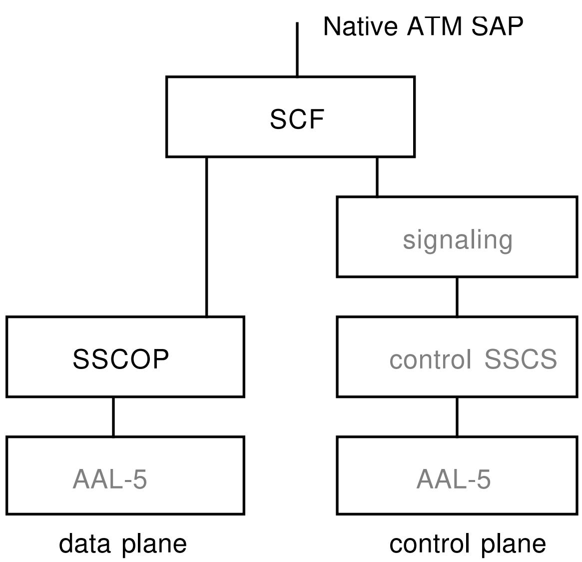
In this case, the data plane protocol stack for Native ATM Services includes the SAR function and CPCS layer (AAL5, AAL1, or a user-defined AAL). The only function that SCF performs is to maintain the state of API_connection and verify that the **ATM_send_data** and **ATM_receive_data** primitives are issued in the appropriate states.



4.3.2 SSCOP_RELIABLE_SSCS

In this case, the data plane protocol stack for Native ATM Services includes the SAR function, the CPCS layer (AAL type 5 only), and the SSCOP protocol. The SCF is defined below. Note that this option is only available for point-to-point connections.

"SSCOP" in this discussion is restricted to an instance of the SSCOP protocol that provides reliability in the data plane; it has nothing to do with the instance of SSCOP that provides a similar service for the signaling protocol. Note that the only SSCOP signals used by SCF are: AA-ESTABLISH, AA-RELEASE, AA-DATA, and MAA-ERROR.



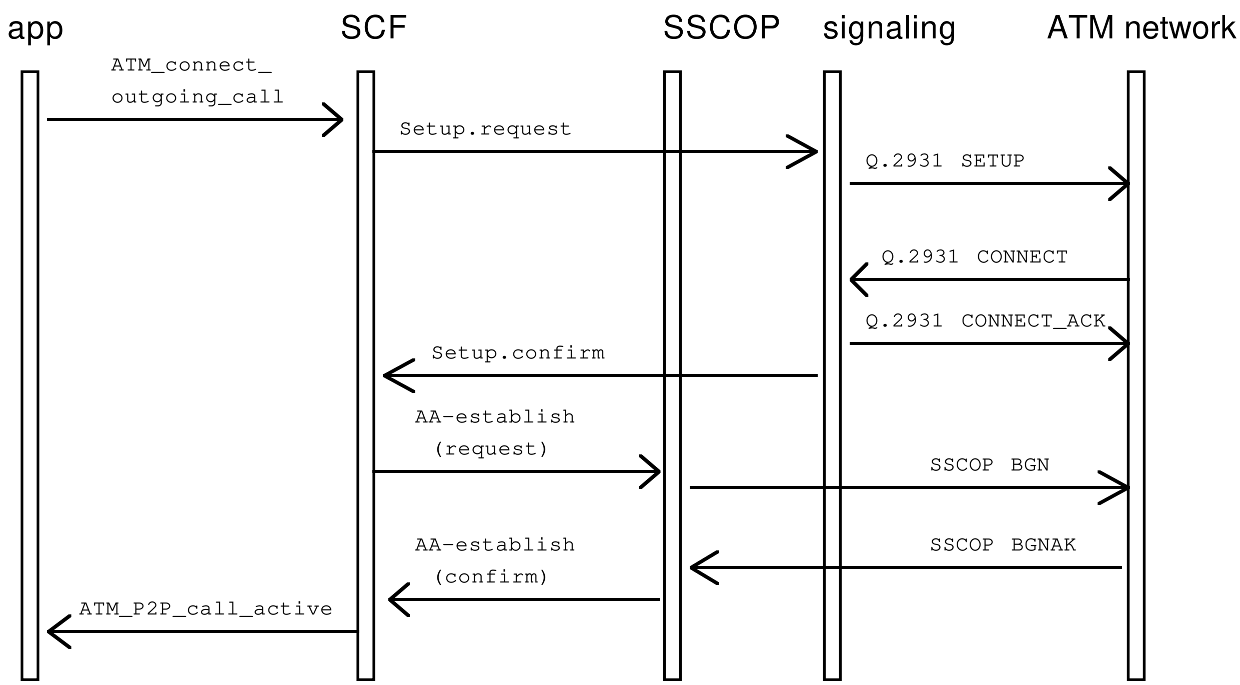
4.3.2.1 Connection Establishment

After **ATM_connect_outgoing_call** is invoked, the SCF in the ATM device initiating the ATM call shall establish an SSCOP connection in the following manner:

1. After the Q.2931 CONNECT message is received from the ATM network, the SCF will invoke the AA-ESTABLISH (request) signal to the SSCOP function. Parameter SSCOP-UU shall be null. Parameter BR shall be YES.

2. The SCF shall wait for an AA-ESTABLISH (confirm) signal from the SSCOP function. Parameter SSCOP-UU shall be ignored.

3. Native ATM Services then indicates that the connection is in state A8, via **ATM_P2P_call_active** primitive.



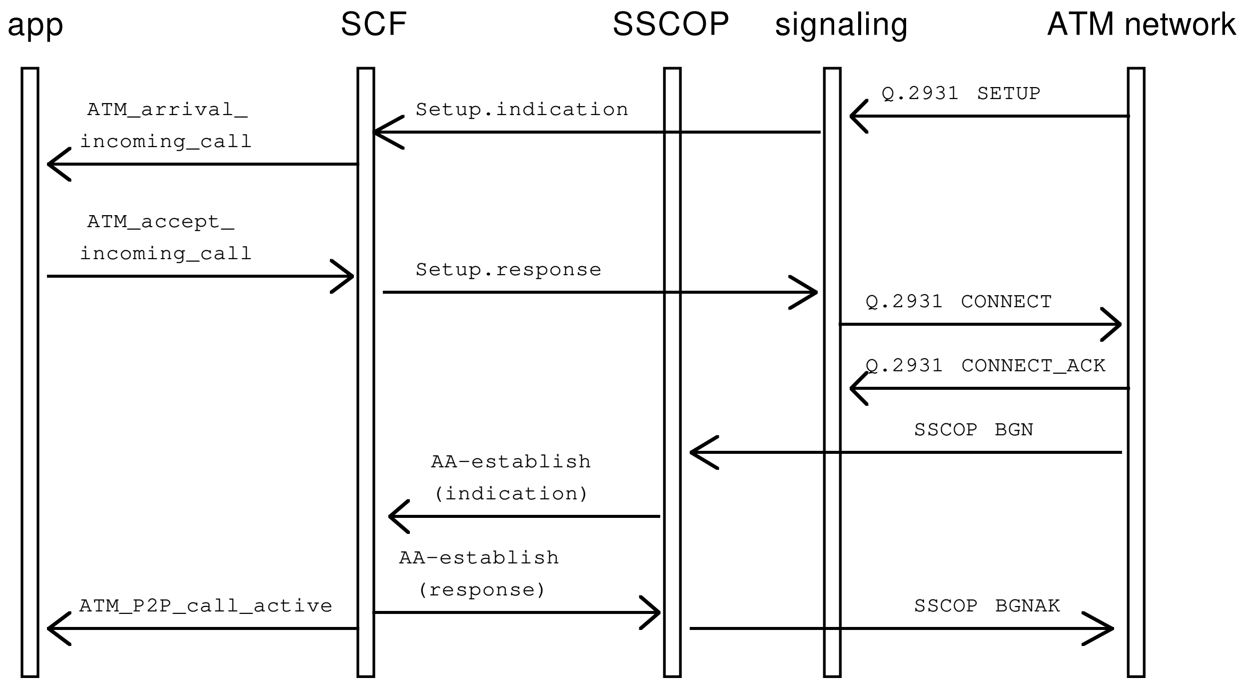
After **ATM_accept_incoming_call** is invoked, the SCF in the ATM device receiving the ATM call shall establish an SSCOP connection in the following manner:

1. After the Q.2931 CONNECT_ACK message is received from the ATM network, the SCF continues to wait.

2. The SCF shall wait for an AA-ESTABLISH (indication) signal from the SSCOP function. Parameter SSCOP-UU shall be ignored.

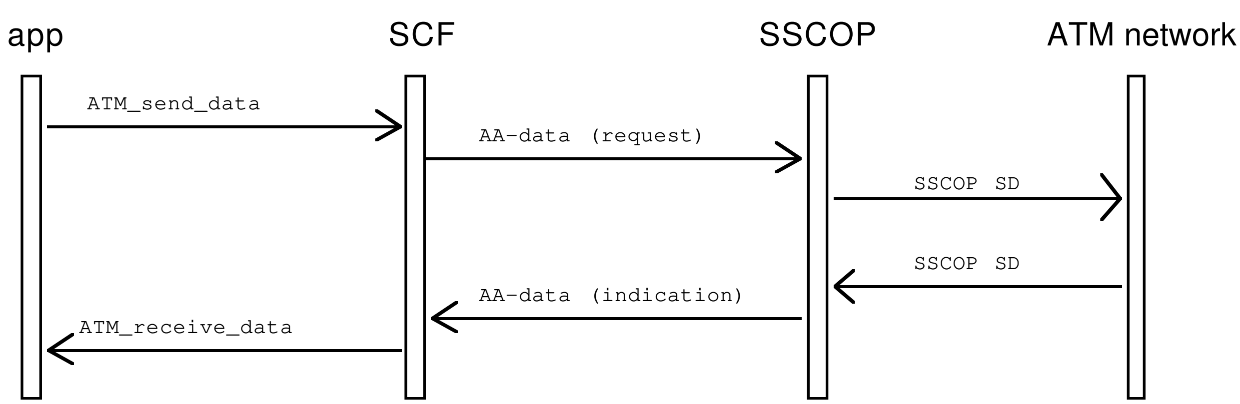
3. The SCF will invoke the AA-ESTABLISH (response) signal to the SSCOP function. Parameter SSCOP-UU shall be null. Parameter BR shall be YES.

4. Native ATM Services then indicates that the connection is in state A8, via **ATM_P2P_call_active** primitive.



4.3.2.2 Data Transfer

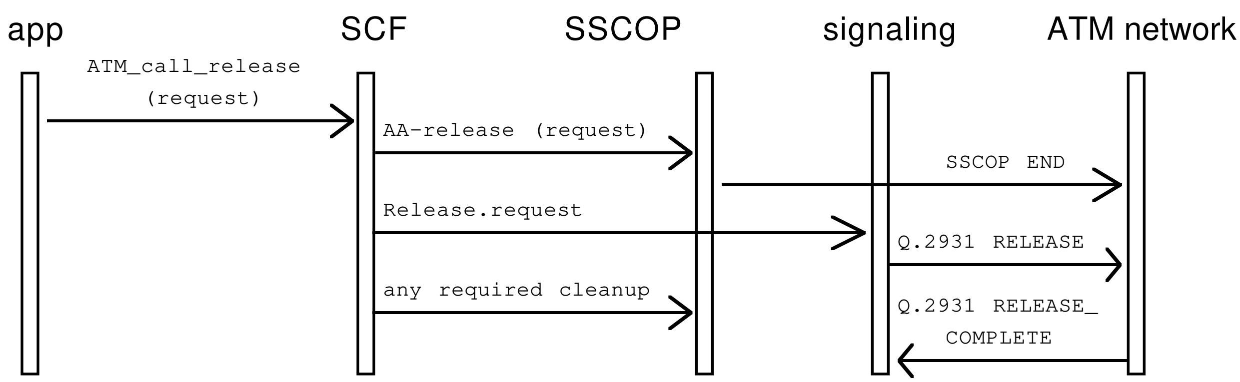
When the **ATM_send_data** primitive is issued, the SCF will invoke an AA-DATA (request) signal to the SSCOP function. When the SSCOP function signals AA-DATA (indication) signal to SCF, then the **ATM_receive_data** primitive is completed. Parameter SN from SSCOP is ignored.



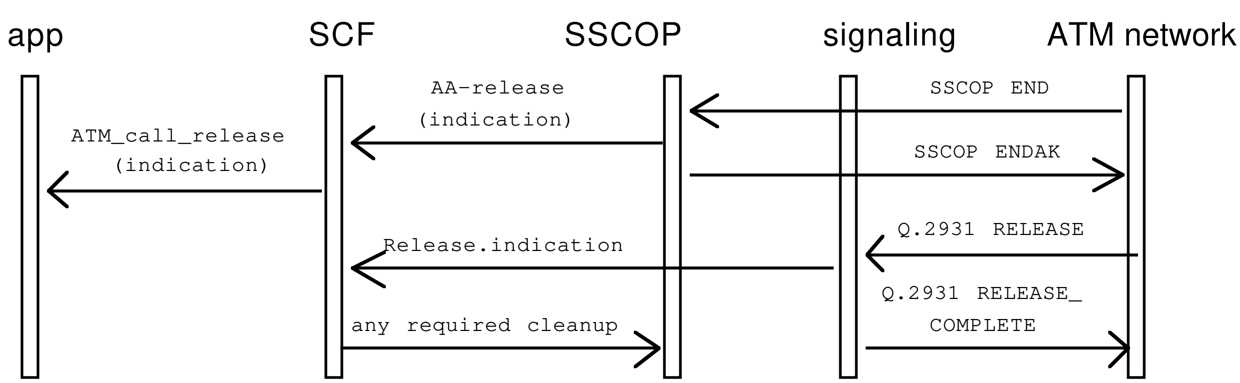
4.3.2.3 Connection Termination

If the **ATM_call_release (request)** primitive is issued, the SCF invokes an AA-RELEASE (request) signal to the SSCOP function. Parameter SSCOP-UU shall be null. After this, the ATM device sends a Q.2931 RELEASE message to the ATM network. If the instantiation of SSCOP requires any cleanup, this should be performed after the SCF has requested the Q.2931 RELEASE message to be sent.

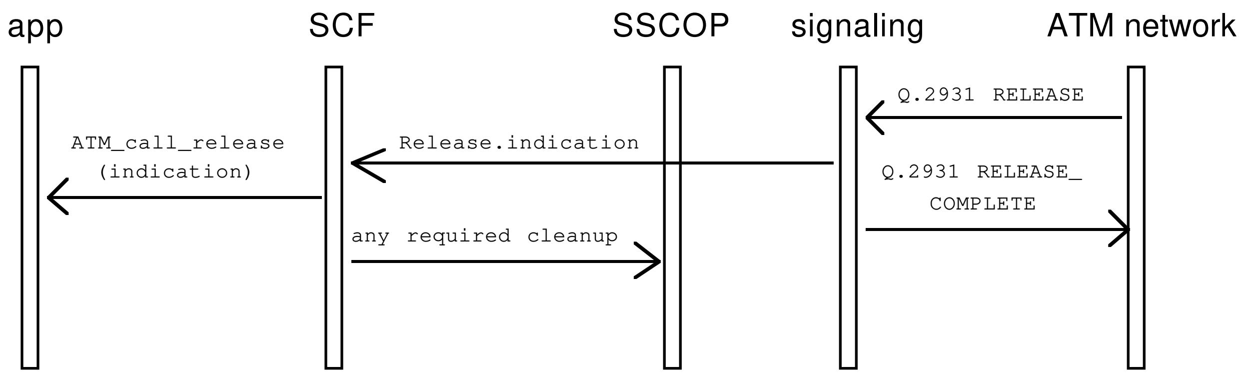
Note that depending on a race condition, an SSCOP ENDAK PDU may or may not arrive across the ATM network before the Q.2931 RELEASE_COMPLETE message. This SSCOP ENDAK PDU should be ignored.



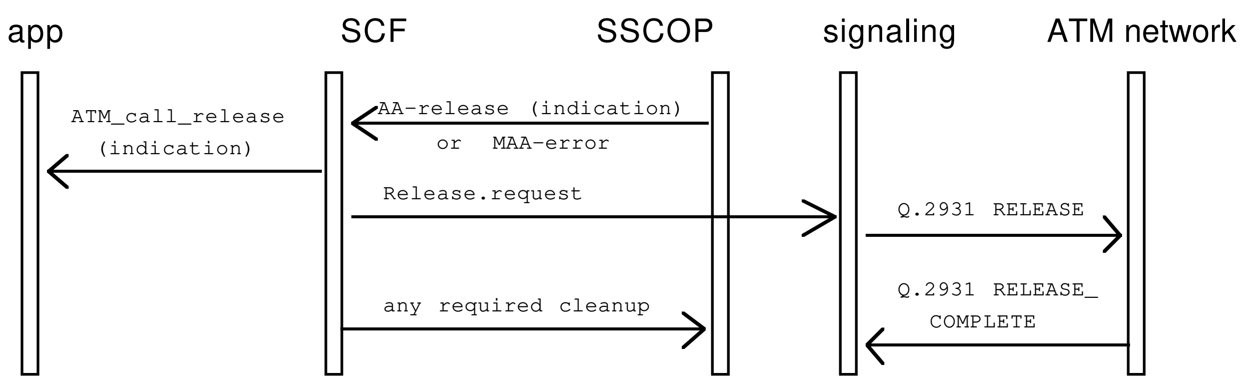
As a result of the previous scenario, the peer SCF on the other end of the ATM connection will receive an AA-RELEASE (indication) signal from the SSCOP function. (The other possibility of the race condition -- a Q.2931 RELEASE being received first -- will be covered below.) When the SCF receives an AA-RELEASE (indication) signal from the SSCOP function, and the SOURCE parameter indicates that the peer user requested the release, then SCF will simply indicate to the application that the connection is no longer available. It is assumed that a Q.2931 RELEASE message will arrive shortly from the ATM network. After the Q.2931 RELEASE message is received, the SCF cleans up any resources from the instantiation of the SSCOP function.



The ATM device may receive an unexpected Q.2931 RELEASE message from the ATM network. This could be the result of the race condition presented in the first flow diagram of this section, or else the ATM network terminates the connection. In these cases, SCF issues a **ATM_call_release (indication)** primitive and cleans up any resources from the instantiation of the SSCOP function.



If the SCF receives an AA-RELEASE (indication) with the SOURCE parameter indicating the local SSCOP has terminated the connection, or MAA-ERROR signal from the SSCOP function; then SCF issues a **ATM_call_release (indication)** primitive and sends a Q.2931 RELEASE message to the ATM network. If the instantiation of SSCOP requires any cleanup, this should be performed after the SCF has requested the Q.2931 RELEASE message to be sent.



4.3.3 SSCOP_UNRELABLE_SSCS

The SCF for this SSCS is for further study.

4.4 Specification of SAP Address

In this document, a "SAP" (service access point) is used for the following purposes:

1. When the application initiates an outgoing call to a remote ATM device, a *destination_SAP* specifies the ATM address of the remote device, plus further addressing that identifies the target software entity within the remote device.

2. When the application prepares to respond to incoming calls from remote ATM devices, a *local_SAP* specifies the ATM address of the device housing the application, plus further addressing that identifies the application within the local device.

NOTE: The preceding explanation refers to a single destination SAP. This is a simplification; Native ATM Services actually supports up to three destination SAPs. For a discussion of this, see section 4.6.

4.4.1 SAP Address as a Vector

The SAP address may be expressed as a vector, (*ATM_addr*, *ATM_selector*, *BLLI_id2*, *BLLI_id3*, *BHLI_id*), where:

- *ATM_addr* corresponds to the 19 most significant octets of a device's 20-octet ATM address (private ATM address structure) or the entire E.164 address (E.164 address structure)
- *ATM_selector* corresponds to the least significant octet of a device's 20-octet ATM address (private ATM address structure only)
- *BLLI_id2* corresponds to a set of octets in the Q.2931 BLLI information element that identify a layer 2 protocol
- *BLLI_id3* corresponds to a set of octets in the Q.2931 BLLI information element that identify a layer 3 protocol
- *BHLI_id* corresponds to a set of octets in the Q.2931 BHLI information element that identify an application (or session layer protocol of an application)

4.4.1.1 SAP Vector Element (SVE)

Each element of the SAP vector is called a SAP Vector Element, or SVE. Each SVE semantically consists of a *SVE_tag*, *SVE_length*, and *SVE_value* field, as defined below. The method used to realize this is implementation specific.

- *SVE_tag* determines the interpretation of the SVE. There are three values defined: PRESENT, ABSENT, and ANY. The value of PRESENT means that the *SVE_value* field contains valid data, which may be compared against corresponding octets found in the Q.2931 SETUP message. The value of ABSENT means that the *SVE_value* field is invalid, and the corresponding octets in the Q.2931 SETUP message are not present. The value of ANY means that the *SVE_value* field is invalid, and that the corresponding octets in the Q.2931 SETUP message may either be not present or assume any value.

  The parameter *destination_SAP* in primitive **ATM_connect_outgoing_call** has these special restrictions:

  1. An *SVE_tag* of ANY is not allowed for any SVE.
  2. An *SVE_tag* of ABSENT is not allowed for the SVE that conveys the ATM address.
  3. For private ATM addresses, the *SVE_tag* for the ATM selector byte is PRESENT; for public ATM addresses, the *SVE_tag* for the ATM selector byte is ABSENT.

- *SVE_length* contains the size, in bytes, of the *SVE_value* field. Note that if the value of *SVE_tag* is ABSENT or ANY, then *SVE_length* shall contain a value of zero.

- *SVE_value*, when valid, contains a value to be compared against corresponding octets found in the Q.2931 SETUP message. For example, the *SVE_value* field for the *ATM_selector* SVE may be compared against the Q.2931 Called Party Number information element. As another example, the *SVE_value* field for the *BHLI_id* SVE may be compared against the Q.2931 BHLI information element.

4.4.2 SVE Encoding

This section specifies the mapping between the *SVE_value* field and the corresponding Q.2931 information element, for each of the five SVE types. See the preceding section for the encoding of the *SVE_tag* and *SVE_length* fields.

In the mappings listed below, the following conventions are used:
- to label bytes of the *SVE_value* field: the first byte is called "1", the second byte is called "2", and so on
- to label octets of the Q.2931 information element: the same label specified in Q.2931 is used.

4.4.2.1 ATM_addr SVE

- Byte 1 of the *SVE_value* field corresponds to octet 5 of the Q.2931 "Called Party Number" information element ("Type of number" and "Addressing/Number plan identification"). Note that this determines whether the ATM address uses private ATM address structure or E.164 address structure.

- When the ATM address uses the private ATM address structure, then bytes 2 - 20 of the *SVE_value* field correspond to octets 6 through the next-to-last octet of the Q.2931 "Called Party Number" information element.

  When the ATM address uses the E.164 address structure, then bytes 2 - N of the *SVE_value* field correspond to octets 6 and beyond of the Q.2931 "Called Party Number" information element.

4.4.2.2 ATM_selector SVE

When the ATM address uses the private ATM address structure, then byte 1 of the *SVE_value* field corresponds to the last octet of the Q.2931 "Called Party Number" information element.

When the ATM address uses the E.164 address structure, then the *SVE_value* field cannot exist; the *SVE_tag* field for the SVE must equal ABSENT or ANY.

4.4.2.3 BLLI_id2 SVE

In this case, there are two options for the structure of the *SVE_value*.

1. Option 1

   - Byte 1 of the *SVE_value* field corresponds to octet 6 of the Q.2931 "Broadband Low Layer Information" information element ("User information layer 2 protocol"). Note that for this option, the value of this byte must not correspond to User Specified (b'10000').

2. Option 2

   - Byte 1 of the *SVE_value* field corresponds to octet 6 of the Q.2931 "Broadband Low Layer Information" information element ("User information layer 2 protocol"). Note that for this option, the value of this byte must correspond to User Specified (b'10000').

   - Byte 2 of the *SVE_value* field corresponds to octet 6a of the Q.2931 "Broadband Low Layer Information" information element ("User specified layer 2 protocol information").

NOTE: This option must be restricted to experimental applications. Uniqueness of these values cannot be guaranteed in a non-experimental ATM environment.

4.4.2.4 BLLI_id3 SVE

In this case, there are five options for the structure of the *SVE_value*.

1. Option 1

   - Byte 1 of the *SVE_value* field corresponds to octet 7 of the Q.2931 "Broadband Low Layer Information" information element ("User information layer 3 protocol"). Note that for this option, the value of this byte must not correspond to ISO/IEC TR 9577 (b'01011') or User Specified (b'10000').

2. Option 2

   - Byte 1 of the *SVE_value* field corresponds to octet 7 of the Q.2931 "Broadband Low Layer Information" information element ("User information layer 3 protocol"). Note that for this option, the value of this byte must correspond to User Specified (b'10000').

   - Byte 2 of the *SVE_value* field corresponds to octet 7a of the Q.2931 "Broadband Low Layer Information" information element ("User specified layer 3 protocol information").

   NOTE: This option must be restricted to experimental applications. Uniqueness of these values cannot be guaranteed in a non-experimental ATM environment.

3. Option 3

   - Byte 1 of the *SVE_value* field corresponds to octet 7 of the Q.2931 "Broadband Low Layer Information" information element ("User information layer 3 protocol"). Note that for this option, the value of this byte must correspond to ISO/IEC TR 9577 (b'01011').

   - Byte 2 of the *SVE_value* field corresponds to octets 7a and 7b of the Q.2931 "Broadband Low Layer Information" information element ("Initial Protocol Identifier"). Note that for this option, the value of this byte must not correspond to IEEE 802.1 SNAP (b'10000000').

4. Option 4

   - Byte 1 of the *SVE_value* field corresponds to octet 7 of the Q.2931 "Broadband Low Layer Information" information element ("User information layer 3 protocol"). Note that for this option, the value of this byte must correspond to ISO/IEC TR 9577 (b'01011').

   - Byte 2 of the *SVE_value* field corresponds to octets 7a and 7b of the Q.2931 "Broadband Low Layer Information" information element ("Initial Protocol Identifier"). Note that for this option, the value of this byte must correspond to IEEE 802.1 SNAP (b'10000000').

   - Bytes 3 - 7 of the *SVE_value* field correspond to octets 8.1- 8.5 of the Q.2931 "Broadband Low Layer Information" information element ("OUI" and "PID").

5. Option 5

   - Byte 1 of the *SVE_value* field corresponds to octet 7 of the Q.2931 "Broadband Low Layer Information" information element ("User information layer 3 protocol"). Note that for this option, the value of this byte must correspond to ISO/IEC TR 9577 (b'01011').

- Byte 2 of the *SVE_value* field corresponds to octets 7a and 7b of the Q.2931 “Broadband Low Layer Information” information element (“Initial Protocol Identifier”) is not present. Therefore, *SVE_length* equals 1.

NOTE: This option is an exception case of BLLI_Id3 that does not identify a layer 3 protocol. The semantic meaning of this option is that each frame in the data plane will contain a header identifying the layer 3 protocol for that frame.

NOTE: This option is equivalent to *SVE_tag* value of ABSENT except for when the vector is used to specify a destination SAP.

4.4.2.5 BHLI_id SVE

In this case, there are three options for the structure of the *SVE_value*.

1. Option 1

- Byte 1 of the *SVE_value* field corresponds to octet 5 of the Q.2931 “Broadband High Layer Information” information element (“High Layer Information Type”). Note that for this option, the value of this byte must correspond to ISO (b’0000000’).

- Bytes 2 - 9 of the *SVE_value* field correspond to octets 6 - 13 of the Q.2931 “Broadband High Layer Information” information element (“High Layer Information”).

2. Option 2

- Byte 1 of the *SVE_value* field corresponds to octet 5 of the Q.2931 “Broadband High Layer Information” information element (“High Layer Information Type”). Note that for this option, the value of this byte must correspond to User Specific (b’0000001’).

- Bytes 2 - 9 of the *SVE_value* field correspond to octets 6 - 13 of the Q.2931 “Broadband High Layer Information” information element (“High Layer Information”).

NOTE: This option must be restricted to experimental applications. Uniqueness of these values cannot be guaranteed in a non-experimental ATM environment.

3. Option 3

- Byte 1 of the *SVE_value* field corresponds to octet 5 of the Q.2931 “Broadband High Layer Information” information element (“High Layer Information Type”). Note that for this option, the value of this byte must correspond to Vendor-Specific (b’0000011’).

- Bytes 2 - 8 of the *SVE_value* field correspond to octets 6 - 12 of the Q.2931 “Broadband High Layer Information” information element (“High Layer Information”).

4.5 Registration of a Local SAP Address

This section specifies the actions taken by Native ATM Services for the purposes of registering a local SAP address by an application through the **ATM_prepare_incoming_call()** primitive.

Native ATM Services must maintain knowledge about all local SAP addresses registered by all applications. When a new SAP is being registered, Native ATM Services must search for a match with all currently registered SAPs and accept the new SAP only when no match is found. If a match is found, Native ATM Services must reject the SAP with a return value of SAP_ALREADY_USED.

A match between two SAPs occurs when all the SVEs match according to the rules in the table below. A "registered vector" may be defined as the result of mapping any SAP address currently registered with Native ATM Services into the structure as specified in section 4.4.1. A "new vector" may be defined as the result of mapping the SAP address which the application is attempting to register into the structure as specified in section 4.4.1.

New SVE	**Registered SVE**		
	SVE_tag = ABSENT	**SVE_tag = PRESENT**	**SVE_tag = ANY**
SVE_tag = ABSENT	Match	No Match	Match
SVE_tag = PRESENT	No Match	Match if Values Match [1]	Match
SVE_tag = ANY	Match	Match	Match

Notes

1. For a more precise specification – the *SVE_value* field of the new vector is compared to the *SVE_value* field of the registered vector; if the values equal each other, then there is a match.

Native ATM Services does not allow multiple overlapping SAPs to be registered at the same time. The use of overlapping SAPs is for further study. However, this does not preclude, wild card SAPs which use the ANY value in SVE_tag field.

NOTE: An implementation may optionally allow a special SAP with ANY as the SVE_tag of every SVE. This SAP violates the above rules, but could be allowed as a special case. This SAP can be registered by only one application on an end system. It receives indication of all calls not distributed to other SAPs (see next section for the description of incoming call distribution).

4.6 Incoming Call Distribution

This section specifies actions taken by native ATM services for the purposes of call distribution to the correct entity at the destination ATM device.

An entity shall be identified by a SAP address. A SAP address shall only identify one entity. An entity may be identified by more than one SAP address. A "registered vector" may be defined as the result of mapping the valid address registered by an entity into the structure as specified in section 4.4.1. An entity shall register a SAP address by issuing the **ATM_prepare_incoming_call** primitive.

An "incoming vector" is defined as the result of mapping the incoming call's Q.2931 information elements into the structure as specified in section 4.4.1. Native ATM Services supports up to three incoming vectors for a single incoming call:

- The "first choice" incoming vector's elements are generated from the following Q.2931 information elements: "Called Party Number", "Broadband High Layer Information", and the first occurrence (if any) of "Broadband Low Layer Information".
- The "second choice" incoming vector's elements are generated from the following Q.2931 information elements: "Called Party Number", "Broadband High Layer Information", and the second occurrence of "Broadband Low Layer Information". If only one occurrence of "Broadband Low Layer Information" is present, then there is no second choice incoming vector.
- The "third choice" incoming vector's elements are generated from the following Q.2931 information elements: "Called Party Number", "Broadband High Layer Information", and the third occurrence of "Broadband Low Layer Information". If no more than two occurrences of "Broadband Low Layer Information" is present, then there is no third choice incoming vector.

When an incoming call arrives, the list of registered vectors is searched for a match with the first choice, then second choice, then third choice incoming vectors. A match between a registered vector and one of the incoming vectors occurs when all of the SVEs match according to the rules in the following table. If a registered vector is found that matches one of the incoming vectors, then the entity corresponding to the vector shall be notified of the incoming call via the **ATM_arrival_of_incoming_call** primitive. If a registered vector is not found, the ATM native services rejects the call with a cause value 88 (incompatible destination).

Incoming SVE	**Registered SVE**		
	SVE_tag = ABSENT	**SVE_tag = PRESENT**	**SVE_tag = ANY**
SVE_tag = ABSENT	Match	No Match	Match
SVE_tag = PRESENT	No Match	Match if Values Match [1]	Match

Notes

1. For a more precise specification – the *SVE_value* field of the incoming vector is compared to the *SVE_value* field of the registered vector; if the two values are equal, then there is a match.

4.7 Compatibility Checking

The ATM Forum Technical Committee User-Network Interface (UNI) Specification Version 3.1 Annex B describes the compatibility checking that is required for an incoming connection.

Call distribution uses some of the parameters that must be checked for compatibility. The parameters of an incoming call will be checked against the parameters registered by applications. If there is a match, the connections will be offered to that application via the *ATM_arrival_of_incoming_call* primitive. If there is no match, the call shall be cleared.

The application shall perform compatibility checking, according to the specified procedure in UNI 3.x, while the connection is in state A6. If this procedure determines that the connection is not compatible, the application shall reject the connection using the *ATM_reject_incoming_call* primitive; otherwise it may accept the connection via the *ATM_accept_incoming_call* primitive.

4.8 Negotiation of Parameters

The ATM Forum Technical Committee User-Network Interface (UNI) Specification Version 3.1 Annex C describes negotiation for Broadband Low Layer Information (BLLI). Annex F describes negotiation for the ATM Adaptation Layer (AAL). These procedures are supported by Native ATM Services.

As part of the BLLI negotiation, the application initiating the call can specify up to three destination SAPs. These SAPs are composed of the following:

- The first choice destination SAP is completely specified by the *destination_SAP* parameter of the **ATM_connect_outgoing_call** primitive.
- The second choice destination SAP contains the ATM_addr SVE, ATM_selector SVE, and BHLI_id SVE from the *destination_SAP* parameter of the **ATM_connect_outgoing_call** primitive. The BLLI_id2 SVE and BLLI_id3 SVE are obtained from the second instance of the connection attributes located in the "Broadband Low Layer Information" information element.
- The third choice destination SAP contains the ATM_addr SVE, ATM_selector SVE, and BHLI_id SVE from the *destination_SAP* parameter of the **ATM_connect_outgoing_call** primitive. The BLLI_id2 SVE and BLLI_id3 SVE are obtained from the third instance of the connection attributes located in the "Broadband Low Layer Information" information element.

4.8.1 Outgoing calls

While the outgoing connection is in state A2, all negotiable parameters may be set using the *ATM_set_connection_attributes* primitive. This includes the setting of up to 3 BLLI choices.

After the connection is in state A8 or A9, the results of negotiation may be obtained via the *ATM_query_connection_attributes* primitive.

4.8.2 Incoming calls

While incoming connection is in state A6, the proposed values for negotiable parameters may be read via the *ATM_query_connection_attributes* primitive. If alternate values are required, they are set via the *ATM_set_connection_attributes* primitive. If nothing is set, the proposed values shall be used for the connection. This includes the first proposed value for BLLI.

5. Reference Documents

The following documents are referred to within this document. They define various aspects of the ATM layers that are being abstracted.

I.361 B-ISDN ATM layer specification

I.362 B-ISDN ATM Adaptation Layer (AAL) Functional Description

I.363 B-ISDN ATM Adaptation Layer (AAL) Specification

Q.2100 B-ISDN Signaling ATM Adaptation Layer Overview Description

Q.2110 B-ISDN - Adaptation Layer - Service Specific Connection Oriented Protocol (SSCOP)

Q.2130 B-ISDN Signaling ATM Adaptation Layer - Service Specific Coordination Function for Support of Signaling at the User-to-Network Interface (SSCF at UNI)

UNI ATM User-Network Interface Specification, version 3.0 and 3.1

ANNEX A: Connection Attributes

A.1 UNI Signaling Related

The items listed in the table below are attributes of connections across ATM networks. In general, these attributes are manipulated using the *ATM_query_connection_attributes* and *ATM_set_connection_attributes* primitives.

Item	Set in State		Q.2931 Encoding		Notes
	A2	A6	Information Element	Octet	
AAL_TYPE	X		ATM Adaptation Layer Parameters	5	
AAL1_SUBTYPE	X			6.1	
AAL1_CBR_RATE	X			7.1	
AAL1_MULTIPLIER	X			8.1 - 8.2	
AAL1_CLOCK_RECOVERY_TYPE	X			9.1	
AAL1_ERROR_CORRECTION	X			10.1	
AAL1_STRUCTURED_DATA_TRANSFER	X			11.1 - 11.2	
AAL1_PARTIALLY_FILLED_CELLS	X			12.1	
AAL5_FWD_MAX_SDU	X	X		6.1 - 6.2	
AAL5_BAK_MAX_SDU	X	X		7.1 - 7.2	
AAL5_SSCS_TYPE	X			8.1	1
USER_DEFINED_AAL_INFO	X	X		6 - 6.3	
FWD_PCR_CLP0	X		ATM Traffic Descriptor	5.1 - 5.3	
FWD_PCR_CLP1	X			7.1 - 7.3	
BAK_PCR_CLP0	X			6.1 - 6.3	
BAK_PCR_CLP1	X			8.1 - 8.3	

Item	Set in State		Q.2931 Encoding		Notes
	A2	A6	Information Element	Octet	
FWD_SCR_CLP0	X		ATM Traffic Descriptor	9.1 - 9.3	
FWD_SCR_CLP1	X			11.1 - 11.3	
BAK_SCR_CLP0	X			10.1 - 10.3	
BAK_SCR_CLP1	X			12.1 - 12.3	
FWD_MBS_CLP0	X			13.1 - 13.3	
FWD_MBS_CLP1	X			15.1 - 15.3	
BAK_MBS_CLP0	X			14.1 - 14.3	
BAK_MBS_CLP1	X			16.1 - 16.3	
BEST_EFFORT	X			17	
FWD_TAGGING	X			18.1	
BAK_TAGGING	X			18.1	
BEARER_CLASS	X		Broadband Bearer Capability	5	
TRAFFIC_TYPE	X			5a	
TIME_REQ	X			5a	
CLIPPING_IND	X			6	
CONNECT_CONFIG	X			6	
APPL_ID_TYPE			Broadband High Layer Information	5	2
APPL_ID				6 - 13	2
BLLI_SELECTOR	X	X	Broadband Low Layer Information		3
LAYER_2_ID	X			6	2, 4
LAYER_2_MODE	X	X		6a	4
LAYER_2_WINDOW_SIZE	X	X		6b	4

Item	Set in State		Q.2931 Encoding		Notes
	A2	A6	Information Element	Octet	
LAYER_2_USER_ID	X		Broadband Low Layer Information	6a	2, 4
LAYER_3_ID	X			7	2, 4
LAYER_3_MODE	X	X		7a	4
LAYER_3_PACKET_SIZE	X	X		7b	4
LAYER_3_WINDOW_SIZE	X	X		7c	4
LAYER_3_USER_ID	X			7a	2, 4
LAYER_3_IPI_ID	X			7a - 7b	2, 4
LAYER_3_OUI_ID	X			8.1 - 8.3	2, 4
LAYER_3_PID_ID	X			8.4 - 8.5	2, 4
CALLED_ADDR_FORMAT			Called Party Number	5	2
CALLED_ADDR				6 etc.	2
CALLED_SUBADDR_TYPE	X		Called Party Subaddress	5	
CALLED_SUBADDR	X			6 etc.	
CALLING_ADDR_FORMAT	X		Calling Party Number	5	
CALLING_ADDR	X			6 etc.	
PRESENTATION_IND	X			5a	
SCREENING_IND	X			5a	
CALLING_SUBADDR_TYPE	X		Calling Party Subaddress	5	
CALLING_SUBADDR	X			6 etc.	
CAUSE_CODING			Cause	2	5, 6
CAUSE_LOCATION				5	5, 6
CAUSE_VALUE				6	5, 6

Item	Set in State		Q.2931 Encoding		Notes
	A2	A6	Information Element	Octet	
CAUSE_DIAGNOSTICS			Cause	7 etc.	5, 6
FWD_QOS_CLASS	X		Quality of Service Parameter	5	
BAK_QOS_CLASS	X			6	
NETWORK_ID	X		Transit Network Selection	6 etc.	

Notes:

1. This connection attribute is encoded at octet 9.1 for UNI 3.0.

2. This connection attribute is set via the ATM_connect_outgoing_call primitive.

3. This connection attribute does not appear in a Q.2931 message; it is used to select the choice of BLLI that other primitives operate on. See primitives **ATM_query_connection_attributes**, **ATM_set_connection_attributes**, **ATM_arrival_of_incoming_call**, and **ATM_accept_incoming_call** for the semantics of this attribute.

4. This connection attribute is conceptually an array of three elements, to support BLLI negotiation.

5. This connection attribute is set via the ATM_abort_connection, ATM_call_release, and ATM_reject_incoming_call primitives.

6. This connection attribute is conceptually an array of two elements, to support two independent suppliers of this information.

A.2 Local Services

Item	Set in State		Values	Notes
	A2	A6		
SSCS (Service Specific Convergence Sublayer)	X	X	NULL, SSCOP_RELIABLE, SSCOP_UNRELIABLE	1

Notes:

1. This selects the services that native ATM services supplies in the data plane for AAL5.

ANNEX B: Management Variables

B.1 UNI Defined

The semantic meaning and codings of these management variables may be found in ATM User Network Specification Chapter 4.

MIB Group and Variable	3.0	3.1	Table Entry
atmfPhysicalGroup			
atmfPortIndex	X	X	X
atmfPortTransmissionType	X	X	X
atmfPortMediaType	X	X	X
atmfPortOperStatus	X	X	X
atmfPortSpecific	X	X	X
atmfPortMyIfName		X	X
atmfMyIpNmAddress		X	
atmfMyOsiNmNsapAddress		X	
atmfAtmLayerGroup			
atmfAtmLayerIndex	X	X	X
atmfAtmLayerMaxVPCs	X	X	X
atmfAtmLayerMaxVCCs	X	X	X
atmfAtmLayerConfiguredVPCs	X	X	X
atmfAtmLayerConfiguredVCCs	X	X	X
atmfAtmLayerMaxVpiBits	X	X	X
atmfAtmLayerMaxVciBits	X	X	X
atmfAtmLayerUniType	X	X	X

MIB Group and Variable	3.0	3.1	Table Entry
atmfAtmLayerUniVersion		X	X
atmfAtmStatsGroup			
atmfAtmStatsIndex	X	X	X
atmfAtmStatsReceivedCells	X	X	X
atmfAtmStatsDroppedReceivedCells	X	X	X
atmfAtmStatsTransmittedCells	X	X	X
atmfVccGroup			
atmfVccPortIndex	X	X	X
atmfVccVpi	X	X	X
atmfVccVci	X	X	X
atmfVccOperStatus	X	X	X
atmfVccTransmitTrafficDescriptorType	X	X	X
atmfVccTransmitTrafficDescriptorParam1	X	X	X
atmfVccTransmitTrafficDescriptorParam2	X	X	X
atmfVccTransmitTrafficDescriptorParam3	X	X	X
atmfVccTransmitTrafficDescriptorParam4	X	X	X
atmfVccTransmitTrafficDescriptorParam5	X	X	X
atmfVccReceiveTrafficDescriptorType	X	X	X
atmfVccReceiveTrafficDescriptorParam1	X	X	X
atmfVccReceiveTrafficDescriptorParam2	X	X	X
atmfVccReceiveTrafficDescriptorParam3	X	X	X
atmfVccReceiveTrafficDescriptorParam4	X	X	X

MIB Group and Variable	3.0	3.1	Table Entry
atmfVccReceiveTrafficDescriptorParam5	X	X	X
atmfVccTransmitQoSClass	X	X	X
atmVccReceiveQoSClass	X	X	X

B.2 Native ATM Services Defined

atmfVccEndpointIdentifier

This element is an extension of atmfVccEntry defined in the atmfVccGroup of UNI 3.0 Chapter 4. Its purpose is to correlate a VCC with an *API_endpoint*.

NOTE: this is a table entry and requires an index parameter.

Annex C: Usage of Wildcards in SAPs

This annex describes the care that must be taken when using “wildcards” to specify SAPs. A “wildcard” is defined as an SVE (SAP vector element) with a tag of ANY. The use of wildcards in a SAP offers flexibility to an application developer. However, this feature shall optionally be used only after the following implications are understood:

- The *destination_SAP* parameter of the **ATM_connect_outgoing_call** primitive may not contain any wildcards.

- When an application successfully registers a SAP with a wildcard, the SAP occupies a large number of the available SAP combinations, based on the matching rules in section 4.5. Thus, careless use of wildcards may preclude other desired applications from registering their SAPs.

- When an application registers a SAP containing a wildcard, the application implies that it supports all specific values for the matching SVE of an incoming vector (including the ABSENT tag) on an incoming call.

- Once an application has accepted an incoming call on an endpoint associated with a wildcard SAP, the application must support and employ the protocols specified by the incoming vector that matched the application’s registered SAP.

Appendix A: Pragmatics of Data Send and Receive

Fundamental issues in sending data are:

1. identifying the location for outgoing data,

2. determining when the source location(s) may be reused, and

3. recognizing/handling sender overflow.

The most common way to specify the data source is to provide the location of a buffer (e.g. via a pointer) and amount of data to send starting from that location. Alternatively, the source data may be specified as an immediate operand to the send operation, i.e., the data is contained on the stack or in a register.

The source buffer cannot be reused until either the data has been deemed sent (or the attempt aborted) or the data has been copied to another buffer. Thus returning from a send operation does not necessarily mean the data has actually been sent. Depending on the communication protocol, (in the case of AAL type 5, depending on assured vs. unassured modes), the data might not be deemed sent until it is known to be successfully received at the destination. Typically, a send operation blocks until the source data is copied or mapped to an intermediate buffer (e.g. in the operating system/network driver). An implementation could also block until the data is deemed sent, but since this incurs latency waiting for the network this usually has poor performance. Yet other possibilities are to poll until the reuse is indicated or to asynchronously interrupt/notify/callback the application to permit buffer reuse.

Sender overflow may occur if the network is not able to send data as fast as an application(s) can transfer data via send operations. In particular, intermediate buffers in the operating system/network driver may overflow. This is an important issue for UBR (unspecified bit rate) traffic: the network capacity devoted to such a connection can change at any time. Thus no amount of sender rate preplanning or buffer size can prevent overflow. Furthermore, the ABR connection capacity can change rapidly. Consequently, implementations may provide a feedback mechanism for indicating and controlling sender overflow. (Of course, a particular implementation is free to simply drop data while an overflow condition exists, but this is not a satisfactory solution for all applications.) Fundamental ways to accomplish this feedback are by blocking (e.g. on a send call), polling (either explicitly or based on some value returned by a send call), or interrupt/notification/callback to the application. It is often effective to tie the source generation rate to the availability of buffers for reuse.

Fundamental issues in receiving data are:

1. identifying the location for depositing incoming data,

2. indicating arrival of data to the application, and

3. recognizing/handling receiver overflow.

Typically, the operating system/network driver chooses a temporary intermediate location in operating system memory for the incoming data. However, it is also possible for the application to specify a location for the incoming data. For example, the application can indicate a memory range that can be transferred to the operating system/network driver as a buffer.

The application can determine the arrival of data via polling or blocking until the data arrives. Alternatively, the application can be informed of data arrival via an interrupt/notification/callback. In polling or blocking, once data has arrived it is transferred via copying or mapping to a location pre-specified by the application in the receive operation. In the case of interrupt/notification/callback, the application can provide a location at the time of such an event to which the data it is transferred via copying or mapping.

Receiver overflow may occur if the application is not able to retrieve (or use data) as fast as the network receives it. As with sender overflow, intermediate buffers in the operating system/network driver may overflow. This is an important issue for UBR (unspecified bit rate) traffic: the network could deliver data at any point; moreover, the application may not be scheduled. Unless the application is guaranteed to attempt to receive data at regular intervals (e.g. via a timer) and the connection PCR (peak cell rate) is set appropriately, it is possible that receiver overflow occurs. Receiver overflow is also an issue for CBR (constant bit rate) connections, unless the application can be guaranteed to be scheduled regularly and consume data. Since the application may not be scheduled and thus not able to consume and thereby replenish the intermediate buffers, there should be an interrupt/notification/callback mechanism for receiver overflow feedback.

For guaranteed connection rates (e.g. CBR), underflow may also be an issue. Sender underflow arises when there is insufficient data available for the network to send. For CBR connections, sender underflow may result in a violation of the guaranteed rate to the destination application. Likewise, receiver underflow arises when there may not be sufficient data available for the application.

Finally, it is very common to use operating system/network driver memory as a intermediate buffer for sending and receiving. Various work has explored eliminating the overhead of this approach by obtaining the source data directly from application memory and storing incoming network data directly into application memory.

Appendix B: Mapping between ATM Forum signaling messages 3.1 and primitives

This annex presents the mapping between the UNI 3.x messages provided by the signaling and the Native ATM Services primitives.

B.1 Primitives invoked by the application

The left column of the following table contains the Native ATM Services primitives and in the right column the resulting signaling messages. The Native ATM Services primitives refer to request or response primitives.

Native ATM Services primitives	ATM Forum signaling messages
ATM_abort_connection	RELEASE
ATM_accept_incoming_call	CONNECT
ATM_add_party	ADD PARTY
ATM_associate_endpoint	None (local significance)
ATM_call_release	RELEASE
ATM_connect_outgoing_call	SETUP
ATM_drop_party	DROP PARTY
ATM_prepare_incoming_call	None (local significance)
ATM_prepare_outgoing_call	None (local significance)
ATM_query_connection_attributes	None (local significance)
ATM_reject_incoming_call	RELEASE
ATM_set_connection_attributes	None (local significance)
ATM_wait_on_incoming_call	None (local significance)

Messages indicated in the ATM Forum signaling message column without correspondence primitive in the Native ATM Services primitives column should not be generated by the application.

B.2 Messages received from Network

The left column of the following table contains the signaling messages and in the right column the resulting Native ATM Services primitives. The Native ATM Services primitives refer to indication or confirm primitives.

ATM Forum signaling messages	Native ATM Services primitives
CALL PROCEEDING	None
CONNECT	ATM_P2P_call_active or ATM_P2MP_call_active
CONNECT ACKNOWLEDGE	ATM_P2P_call_active or ATM_P2MP_call_active
SETUP	ATM_arrival_of_incoming_call
RELEASE	ATM_call_release
RELEASE COMPLETE	None
STATUS	None [1]
STATUS INQUIRY	None
RESTART	ATM_call_release
RESTART ACKNOWLEDGE	None
ADD PARTY	Invalid under the restrictions in this specification
ADD PARTY ACKNOWLEDGE	ATM_add_party_success
ADD PARTY REJECT	ATM_add_party_reject
DROP PARTY	ATM_drop_party
DROP PARTY ACKNOWLEDGE	None

1 This depends on the cause information element. In some cases, this message may imply the release of the connection.

Native ATM Services: Semantic Description

Signaling messages that correspond to ‘None’ (in the Native ATM Services primitives column) do not generate a primitive that reaches the application.

Appendix C: Example SAP Combinations

This appendix provides examples of users of Native ATM Services can be specify SAPs. This is not meant to restrict implementations, but rather to provide helpful guidance.

C.1 SAPs for Data Link Layer Protocols

Characteristics

This group of SAPs is used when the destination of the connection is a Data Link Layer protocol entity. This provides support for the transport of several network protocols over a single VC, which may be cost effective when the cost of a VC is a consideration. Examples of these data link protocols include LLC (Logical Link Control) and PPP (Point-to-Point Protocol).

Semantic Definition

The SAP address consists of the ATM address, including the selector byte, and the identification of a Layer 2 protocol

Specific Coding

The Layer 2 protocol specification is encoded in the following way:

ATM_addr.SVE_tag = PRESENT

ATM_addr.SVE_value = the ATM address, minus the selector byte

ATM_selector.SVE_tag = PRESENT (private ATM address) or ABSENT (E.164 address)

ATM_selector.SVE_value = the selector byte (from the private ATM address)

BLLI_id2.SVE_tag = PRESENT

BLLI_id2.SVE_value = the identification of the layer 2 protocol

BLLI_id3.SVE_tag = ABSENT

BHLI_id.SVE_tag = ABSENT

C.2 SAPs for Network Layer Protocols

Characteristics

This group of SAPs is used when the destination of the connection is a Network or Transport Layer protocol entity, or a service transport entity. This provides support for a single network or transport protocol, or a single service transport over a single VC. Examples of these network protocols include X.25 and IP. Examples of transport protocols are TCP and TP4. Examples of service transport entities are ATM LAN emulation and ATM Circuit emulation.

Semantic Definition

The SAP address consists of the ATM address, including the selector byte, and the specification of a Layer 3 protocol.

Specific Coding

The Layer 3 protocol specification is encoded in the following way:

ATM_addr.SVE_tag = PRESENT

ATM_addr.SVE_value = the ATM address, minus the selector byte

ATM_selector.SVE_tag = PRESENT (private ATM address) or ABSENT (E.164 address)

ATM_selector.SVE_value = the selector byte (from the private ATM address)

BLLI_id2.SVE_tag = ANY

BLLI_id3.SVE_tag = PRESENT

BLLI_id3.SVE_value = the identification of the layer 3 protocol (options 1 - 3 only)

BHLI_id.SVE_tag = ABSENT

C.3 SAPs for Higher Layer Protocols and ATM-Aware Applications

Characteristics

This group of SAPs is used when the destination of the connection is an ATM-aware application, or more properly, the session layer of an application. This provides for the transport of a single application's data over a single VC.

Semantic Definition

The SAP address consists of the ATM address, including the selector byte, identification of the layer-2 protocol used (if present), identification of the layer-3 protocol used (if present), and the identification of the higher layer protocol or an ATM-aware application.

Specific Coding

The ATM-aware application specification is encoded in the BHLI information element.

ATM_addr.SVE_tag = PRESENT

ATM_addr.SVE_value = the ATM address, minus the selector byte

ATM_selector.SVE_tag = PRESENT (private ATM address) or ABSENT (E.164 address)

ATM_selector.SVE_value = the selector byte (from the private ATM address)

BLLI_id2.SVE_tag = ABSENT or PRESENT

BLLI_id2.SVE_value = identification of the layer 2 protocol (if tag = PRESENT)

BLLI_id3.SVE_tag = ABSENT or PRESENT

BLLI_id3.SVE_value = identification of the layer 3 protocol (if tag = PRESENT)

BHLI_id.SVE_tag = PRESENT

BHLI_id.SVE_value = the identification of the ATM-aware application