Ada Binding Reference Manual - SAFENET Lightweight Application Services

Bert J. Dempsey, John C. Fenton, Jeffrey R. Michel,
Alexander S.Waterman and Alfred Weaver
Computer Networks Laboratory
University of Virginia

1. Introduction

This manual describes the Ada binding to the SAFENET Lightweight Application Services. The binding consists of several Ada package specifications which contain data types and subprograms that provide the lightweight service primitives described in chapter four of reference [1]. This manual describes the services provided by each Ada package in terms of the data types and subprograms defined therein. Portions of this report were prepared while one of the authors (Jeff Michel) was in residence at the Software Engineering Institute in July 1992.

2. Application Program Interface Semantics

The binding consists of the following Ada packages.

- LW_ADDRESS_MANAGEMENT
- LW_COMMUNICATIONS_SUPPORT
- LW_CONNECTION_MANAGEMENT
- LW_DATA_TRANSFER
- LW_ERROR_MANAGEMENT
- LW_PROTOCOL_MANAGEMENT
- LW_TRANSACTION_SERVICES

The following subsections describe the abstractions provided by the data types declared in each package and the binding primitives provided by the subprograms. For each primitive its semantics, including the exceptions it may raise, and the data structures it uses are discussed.

2.1. LW_ADDRESS_MANAGEMENT

The primitives in this package provide directory services to an application program. They allow an application to bind logical names to physical addresses, remove address bindings, look up address bindings, and discover the number of address bindings that exist. The Ada binding does not specify whether address bindings are local to an application program or distributed among several application programs that also use the SLA suite. However, in the current implementation of the Ada binding, address bindings are local to an application program.[1]

```
with LW_COMMUNICATIONS_SUPPORT;

package LW_ADDRESS_MANAGEMENT is
   package LWCS renames LW_COMMUNICATIONS_SUPPORT;

   type ADDRESS_ID is private;
```

An `ADDRESS_ID` denotes a specific and existing logical-name-to-physical-address binding. It may be used to refer to an address binding in a call to another address management primitive, or it may be used to succinctly specify the network and transport address of a communication endpoint.

```
   type PHYSICAL_ADDRESS is new LWCS.BYTE_BUFFER;
   pragma PACK (PHYSICAL_ADDRESS);
```

The `PHYSICAL_ADDRESS` type may contain the FDDI MAC or ISO network address (NSAP) of a given host computer or may contain the address of a transport service access point (TSAP) within a given host. This type is defined simply in terms of bytes, so that no address format is inherent in the definition. However, the length of the physical address must conform to that of the address type it contains if the physical address is to be valid. In accordance with chapter 5 of reference [2], a MAC address must be six bytes long (section 5.2.6), a network address must be 20 bytes long (section 5.2.4), and a transport address must be four bytes long (section 5.2.3.3).

1. Future plans exist to provide an implementation in which distributed address binding is supported.

```
procedure BIND_ADDRESS
  ( NAME                    : in      LWCS.LOGICAL_NAME;
    NETWORK                 : in      PHYSICAL_ADDRESS;
    TRANSPORT               : in      PHYSICAL_ADDRESS;
    ID                      :    out  ADDRESS_ID);
```

The BIND_ADDRESS primitive binds an unbound logical name, NAME, to the ordered pair (NETWORK, TRANSPORT) of physical addresses. It provides an address identifier ID which can be used to identify the address binding in other LW_ADDRESS_MANAGEMENT primitives or to provide an address of a communication endpoint. It is permissible to have more than one logical name bound to one ordered pair of physical addresses. It is not permissible, however, to bind one logical name to more than one (NETWORK, TRANSPORT) address pair. That is, if NAME is already bound, this call raises the exception LOGICAL_NAME_ALREADY_BOUND_ERROR. If the network or transport address is not a valid address, the call raises the exception INVALID_NETWORK_ADDRESS_ERROR or INVALID_TRANSPORT_ADDRESS_ERROR, respectively. In the case of an exception, the value of ID is undefined.

```
procedure UNBIND_ADDRESS
  ( NAME                    : in      LWCS.LOGICAL_NAME);

procedure UNBIND_ADDRESS
  ( ID                      : in      ADDRESS_ID);
```

The UNBIND_ADDRESS primitive removes address bindings. It is overloaded to remove bindings specified by either logical name, NAME, or address identifier, ID. If NAME is not bound to an address, or the binding referred to by ID no longer exists, the primitive raises the exception NONEXISTENT_BINDING_ERROR.

```
procedure LOOKUP
  ( NAME                     : in      LWCS.LOGICAL_NAME;
    NETWORK                  :    out  PHYSICAL_ADDRESS;
    TRANSPORT                :    out  PHYSICAL_ADDRESS;
    ID                       :    out  ADDRESS_ID);

procedure LOOKUP
  ( ID                       : in      ADDRESS_ID;
    NETWORK                  :    out  PHYSICAL_ADDRESS;
    TRANSPORT                :    out  PHYSICAL_ADDRESS;
    NAME                     :    out  LWCS.LOGICAL_NAME);
```

The LOOKUP primitive returns bindings. It is overloaded to search for bindings given either a logical name NAME, or an address identifier ID. In either case, the NETWORK and TRANSPORT parameters are set to the corresponding addresses of the address binding. When the search is based on a logical name, the call provides the address identifier ID associated with the binding. Conversely, when the search is based on the address identifier ID the call provides the logical name NAME. If NAME is not bound to an address, or the binding referred to by ID no longer exists, the primitive raises the exception NONEXISTENT_BINDING_ERROR. If the network or transport address is not a valid address, the call raises the exception INVALID_NETWORK_ADDRESS_ERROR or INVALID_TRANSPORT_ADDRESS_ERROR, respectively. In the case of an exception, the values of the **out** parameters are undefined.

```
    function NUMBER_OF_ADDRESSES_BOUND
       return NATURAL;

end LW_ADDRESS_MANAGEMENT;
```

The NUMBER_OF_ADDRESSES_BOUND primitive returns the number of address bindings that presently exist.

Note that all address bindings are made by the application program after its execution. Bindings do not persist after termination of the application program.

2.2. LW_COMMUNICATIONS_SUPPORT

The primitives and data types declared in this package provide fundamental data structures and operations on activities. Activities are discussed below.

```
with SYSTEM;

package LW_COMMUNICATIONS_SUPPORT is

   LOGICAL_NAME_LENGTH : constant := 255;
   subtype LOGICAL_NAME is STRING (1 .. LOGICAL_NAME_LENGTH);
```

A `LOGICAL_NAME` is a fixed-length character string which provides a logical name for the physical address of a communication endpoint. This physical address is the pair of network and transport addresses of the communication endpoint.

```
   type UNSIGNED_BYTE is range 0 .. 255;
   for UNSIGNED_BYTE'SIZE use 8;

   type BYTE_BUFFER is array (INTEGER range <>) of UNSIGNED_BYTE;
   pragma PACK (BYTE_BUFFER);

   MAX_INITIAL_DATA_BUFFER_SIZE : constant := 65535 - 76;

   MAX_DATA_BUFFER_SIZE : constant := 65535;
   subtype DATA_BUFFER_SIZE is INTEGER range 1 .. MAX_DATA_BUFFER_SIZE;
   type DATA_BUFFER is array (DATA_BUFFER_SIZE range <>) of UNSIGNED_BYTE;
   pragma PACK (DATA_BUFFER);

   INITIAL_DATA_BUFFER_OVERHEAD : constant := 76;
   MAX_INITIAL_DATA_BUFFER_SIZE : constant := MAX_DATA_BUFFER_SIZE -
     INITIAL_DATA_BUFFER_OVERHEAD;
   subtype INITIAL_DATA_BUFFER_SIZE is INTEGER range
     1 .. MAX_INITIAL_DATA_BUFFER_SIZE;
   type INITIAL_DATA_BUFFER is array (INITIAL_DATA_BUFFER_SIZE range <>) of
     UNSIGNED_BYTE;
   pragma PACK (INITIAL_DATA_BUFFER);
```

The `DATA_BUFFER` may be used to hold data in primitives which do not require an address. In contrast, the `INITIAL_DATA_BUFFER` may be used to hold data in primitives in which the addresses of the sending or receiving endpoints are provided. This type of data buffer has a slightly smaller maximum length than `DATA_BUFFER` due to the addressing overhead, `INITIAL_DATA_BUFFER_OVERHEAD`, associated with the initial data.

```
type MESSAGE_PRIORITY is range 0 .. 255;
for MESSAGE_PRIORITY'SIZE use 32;
```

The MESSAGE_PRIORITY indicates the priority of the data sent by a sending primitive. Higher numerical values indicate higher priority. The semantics are such that data of higher priority is processed by lower layer protocols before data of lower priority.

```
subtype ACTIVITY_INDEX is NATURAL;

type ACTIVITY_MODE is (ASYNCHRONOUS, SYNCHRONOUS);

type ACTIVITY_STATE is (ERROR, IN_PROGRESS, NONEXISTENT, SUCCESS);

type ACTIVITY_BLOCK is
  record
    BYTES_RECEIVED : NATURAL;
    END_OF_MESSAGE : BOOLEAN;
    MODE : ACTIVITY_MODE;
    PRIORITY : MESSAGE_PRIORITY;
  end record;

type ACTIVITY_BLOCK_POINTER is private;
```

The above types are all associated with the idea of an “activity”. An activity represents the execution of a binding primitive. An ACTIVITY_INDEX is a handle on an activity which is being performed asynchronously. Such indices are returned by primitives which are executed asynchronously and may be used to refer to the primitive’s execution in calls to certain primitives described below. The ACTIVITY_MODE allows the execution of a primitive to be specified as asynchronous or synchronous. An activity may be in any of the states defined by the type ACTIVITY_STATE. See Appendix A for a description of the activity state model. An ACTIVITY_BLOCK_POINTER provides a reference to an activity block so that the components of the block may be accessed when an asynchronous activity completes.

The ACTIVITY_BLOCK indicates scheduling parameters for an activity. The meanings of its components are as follows. When the MODE component is set to ASYNCHRONOUS, the calling Ada task[1] does not block waiting for the communication primitive's request to finish. Instead the subprogram call may complete before the activity completes. The only way an application may determine if an activity has completed is via a call to the GET_ACTIVITY_STATE or WAIT_ON_ACTIVITY primitives declared in this package. When the MODE component is set to SYNCHRONOUS, the task blocks for an unbounded amount of time until the communication primitive's activity completes[2]. Note that an activity may complete either successfully or in error. Primitives which accept neither an activity block nor an activity block pointer are performed synchronously.

The values of the BYTES_RECEIVED, END_OF_MESSAGE and PRIORITY components of the activity block have different meanings depending on whether they are used in primitives which send data or receive it. For sending primitives the meanings are as follows.

In a sending primitive the END_OF_MESSAGE, PRIORITY, and BYTES_RECEIVED components have the following semantics. The END_OF_MESSAGE component applies in the following way to the connection-oriented primitives SEND_MESSAGE and GET_MESSAGE, declared in the package LW_DATA_TRANSFER. When set TRUE, END_OF_MESSAGE serves to mark a message boundary at the end of the byte stream of the message sent to the destination endpoint. When such a message arrives at the destination endpoint, the receiving primitive completes its execution, immediately returning the data it has received up to the message boundary even if the buffer given in the receiving primitive is not yet full. If END_OF_MESSAGE is set FALSE, the sending primitive does not mark a message boundary at the end of the message sent to the destination endpoint. When such a message arrives at the destination endpoint, the

1. Unless stated otherwise, semantics that apply to an Ada task also apply to the main program's thread of control. For brevity, the discussion is in terms of tasks.
2. A provision for bounding blocking is provided in Appendix A of reference [3].

receiving primitive will not deliver the data sent in the sending primitive until either the receiver's data buffer is full, or the connection closes. For sending unitdata and transaction primitives, the END_OF_MESSAGE component has a no semantic effect.

The PRIORITY component sets the priority of the message sent in a primitive. The BYTES_RECEIVED component has no effect upon the semantics of sending primitives.

For receiving primitives, the meanings of the activity block components BYTES_RECEIVED, END_OF_MESSAGE and PRIORITY are as follows. It is crucial to note that for asynchronous calls to receive primitives, the components of an activity block are set asynchronously when the activity completes. For this reason the caller supplies a pointer to an activity block in the call, rather than the activity block itself. Because the activity block components are set asynchronously, there are several conditions that must be met; otherwise, the meanings of the values of these components are undefined. These conditions apply to both synchronous and asynchronous calls. The conditions are:

- The activity block referenced by the pointer given in the call to a receiving primitive must not have been given as an actual parameter to any sending primitive which could possibly execute during the execution of the receiving primitive.
- The activity block referenced by the pointer given in the call to a receiving primitive must not be referenced by the pointer given to any other receiving primitive which could possibly execute during the execution of the receiving primitive.
- The full execution of the primitive to which the activity block is bound must have completed successfully. For synchronous calls this is only the case if no exception is raised at the point of the call. For asynchronous calls this is only the case when either a call to the function GET_ACTIVITY_STATE for the activity index provided by the receiving primitive's call yields the result SUCCESS, or a call to the procedure WAIT_ON_ACTIVITY does not raise an exception.

When the above conditions are met, the BYTES_RECEIVED component holds the number of bytes placed in the data buffer given to the receiving primitive. The PRIORITY component indicates the priority of the message which caused the data to be delivered to the application. The message which caused delivery did so because either it (1) was a unitdata message, a transaction request or a transaction response, or (2) its arrival caused the receiver's data buffer to fill, or (3) it had a message boundary marked at its end. For the GET_MESSAGE primitive, the END_OF_MESSAGE component is set FALSE if a message boundary caused the delivery of data to the application, and FALSE otherwise. In receiving unitdata and transaction primitives, the END_OF_MESSAGE component is set TRUE if the receive buffer was large enough to contain the entire received message, and FALSE otherwise.

```
function GET_ACTIVITY_BLOCK_POINTER
  ( BLOCK                  : in      ACTIVITY_BLOCK)
    return ACTIVITY_BLOCK_POINTER;
```

The GET_ACTIVITY_BLOCK_POINTER primitive takes an activity block, BLOCK, as its single argument and returns a pointer to the block. This primitive must be used to provide a reference to an activity block for primitives which may receive data. As mentioned above, these primitives need the address of the activity block so that, in the case of asynchronous calls, they may asynchronously change components of the activity block when the activity completes.

```
function GET_ACTIVITY_STATE
  ( ID                     : in      ACTIVITY_INDEX)
    return ACTIVITY_STATE;
```

The GET_ACTIVITY_STATE primitive returns the state of the activity referred to by the identifier, ID. One may query the state of a *pending* activity an unlimited number of times to discover when it completes. However, once this call returns an activity state indicating success, the activity will be unbound from its primitive and shall become NONEXISTENT. If the call returns the result ERROR, the activity shall not be unbound from its primitive until a call to the GET_ERROR primitive is made. In the case of a call to GET_ACTIVITY_STATE with an activity index which is not bound to a primitive, GET_ACTIVITY_STATE returns the result NONEXISTENT. Note that synchronous activities always return an activity index of zero, and this activity index is never bound to any primitive. Hence a query of the state of a synchronous primitive call shall always give the result NONEXISTENT.

```
      procedure WAIT_ON_ACTIVITY
        ( ID                       : in      ACTIVITY_INDEX);

   end LW_COMMUNICATIONS_SUPPORT;
```

The WAIT_ON_ACTIVITY primitive blocks its calling task until the execution of the activity referred to by the identifier, ID, completes, placing the activity in the UNKNOWN state. If the activity completes in error, the primitive raises the exception corresponding to the error encountered. If ID is not bound to an activity, the primitive raises the exception ACTIVITY_UNKNOWN_ERROR.

2.3. LW_CONNECTION_MANAGEMENT

The primitives in this package provide support for applications to create and destroy connections, modify their options, and check their state.

```
   with LW_ADDRESS_MANAGEMENT;
   with LW_COMMUNICATIONS_SUPPORT;
   with LW_PROTOCOL_MANAGEMENT;
   with SYSTEM;

   package LW_CONNECTION_MANAGEMENT is
      package LWAM renames LW_ADDRESS_MANAGEMENT;
      package LWCS renames LW_COMMUNICATIONS_SUPPORT;
      package LWPM renames LW_PROTOCOL_MANAGEMENT;

      type CONNECTION_ID is private;
```

The CONNECTION_ID is a handle on a connection. A valid CONNECTION_ID may only be obtained from an OPEN_CONNECTION, OPEN_CONNECTION_WITH_DATA, ACCEPT_CONNECTION or ACCEPT_ CONNECTION_WITH_DATA primitive.

```
type CONNECTION_STATE is (ACCEPT_REQUEST_SENT, GRACEFUL_CLOSE_SENT,
  NONEXISTENT, OPEN, OPEN_REQUEST_SENT, TIMED_OUT);
```

The CONNECTION_STATE type represents the state of a particular connection. The connection state model is described in Appendix B.

```
type TERMINATE_MODE is (GRACEFUL, IMMEDIATE);
```

The TERMINATE_MODE type allows one to specify graceful or immediate semantics for the close of a connection.

```
procedure OPEN_CONNECTION
  ( DESTINATION            : in      LWAM.ADDRESS_ID;
    SOURCE                 : in      LWAM.ADDRESS_ID;
    ACTIVITY_PARAMETERS    : in      ACTIVITY_BLOCK;
    PARAMETERS             : in      LWPM.CONNECTION_PARAMETERS;
    OPTIONS                : in      LWPM.MESSAGE_OPTIONS;
    CONNECTION             :    out  CONNECTION_ID;
    INDEX                  :    out  LWCS.ACTIVITY_INDEX);
```

The OPEN_CONNECTION primitive initiates a connection to the remote endpoint referred to by the address identifier, DESTINATION, from the endpoint on the local host referred to by the address identifier, SOURCE. The ACTIVITY_PARAMETERS block provides the additional semantics for sending primitives described in section 2.2. The PARAMETERS parameter provides connection parameters as described in section 2.6. The OPTIONS parameter provides message options as described in section 2.6. The primitive provides a unique connection identifier, CONNECTION, which provides a way to refer to the connection in other binding primitives. The primitive also provides an activity index, INDEX. If either DESTINATION or SOURCE are invalid, the primitive raises the exception INVALID_ADDRESS_ID_ERROR. If either the SOURCE or DESTINATION addresses are already in use, or the SOURCE address is not multicast and refers to an endpoint which is not on the local host, the primitive raises the exception INVALID_OPERATION_ERROR. If the underlying communications protocol is unable to allocate resources for the new connection, the primitive raises the exception UNABLE_TO_ALLOCATE_PROTOCOL_RESOURCES_ERROR. If the DESTINATION endpoint fails to accept the connection before an implementation-defined timeout expires, the primitive raises the exception COMMUNICATION_ATTEMPT_TIMED_OUT_ERROR. In the case of an exception, the **out** parameters are undefined.

```
procedure OPEN_CONNECTION_WITH_DATA
  ( DESTINATION            : in      LWAM.ADDRESS_ID;
    SOURCE                 : in      LWAM.ADDRESS_ID;
    ACTIVITY_PARAMETERS    : in      ACTIVITY_BLOCK;
    DATA                   : in      SYSTEM.ADDRESS;
    LENGTH                 : in      LWCS.INITIAL_DATA_BUFFER_SIZE;
    PARAMETERS             : in      LWPM.CONNECTION_PARAMETERS;
    OPTIONS                : in      LWPM.MESSAGE_OPTIONS;
    CONNECTION             :    out  CONNECTION_ID;
    INDEX                  :    out  LWCS.ACTIVITY_INDEX);
```

The OPEN_CONNECTION_WITH_DATA primitive inherits all the behavior of the OPEN_CONNECTION primitive and also includes the ability to transmit an initial data buffer as part of the connection setup procedure. The DATA parameter provides the base address of the contiguous data buffer to be transferred, and the LENGTH parameter specifies the length, in bytes, of the buffer. If any part of the data buffer is not at a valid memory address, the primitive raises the exception INVALID_PARAMETERS_ERROR. In the case of an exception, the **out** parameters are undefined.

```
procedure ACCEPT_CONNECTION
  ( RECEIVER               : in      LWAM.ADDRESS_ID;
    ACTIVITY_PARAMETERS    : in      ACTIVITY_BLOCK_POINTER;
    PARAMETERS             : in      LWPM.CONNECTION_PARAMETERS;
    OPTIONS                : in      LWPM.MESSAGE_OPTIONS;
    CONNECTION             :    out  CONNECTION_ID;
    INDEX                  :    out  LWCS.ACTIVITY_INDEX);
```

The ACCEPT_CONNECTION primitive awaits a connection to the local address referred to by the address identifier, RECEIVER. The ACTIVITY_PARAMETERS block provides the additional semantics for receiving primitives described in section 2.2. The PARAMETERS parameter provides connection parameters as described in section 2.6. The OPTIONS parameter provides message options as described in section 2.6. The primitive provides a unique connection identifier, CONNECTION, which provides a way to refer to the connection in other binding primitives. The primitive also provides an activity index,

INDEX. If RECEIVER is not valid, the primitive raises the exception INVALID_ADDRESS_ID_ERROR. If the RECEIVER address is already in use or is not multicast and refers to a remote host, the primitive raises the exception INVALID_OPERATION_ERROR. If the underlying communications protocol is unable to allocate resources for the new connection, the primitive raises the exception UNABLE_TO_ALLOCATE_PROTOCOL_RESOURCES_ERROR. In the case of an exception, the **out** parameters are undefined.

```
procedure ACCEPT_CONNECTION_WITH_DATA
  ( RECEIVER               : in      LWAM.ADDRESS_ID;
    ACTIVITY_PARAMETERS    : in      ACTIVITY_BLOCK_POINTER;
    DATA                   : in      SYSTEM.ADDRESS;
    BUFFER_SIZE            : in      LWCS.DATA_BUFFER_SIZE;
    PARAMETERS             : in      LWPM.CONNECTION_PARAMETERS;
    OPTIONS                : in      LWPM.MESSAGE_OPTIONS;
    CONNECTION             :    out  CONNECTION_ID;
    INDEX                  :    out  LWCS.ACTIVITY_INDEX);
```

The ACCEPT_CONNECTION_WITH_DATA primitive inherits all the behavior of the ACCEPT_CONNECTION primitive and also includes the ability to receive data as part of the connection setup procedure. The DATA parameter provides the base address of the contiguous data buffer which is to hold received data, and the BUFFER_SIZE parameter specifies the size, in bytes, of the buffer. If any part of the data buffer is not at a valid memory address and is used to receive data, the primitive raises the exception INVALID_PARAMETERS_ERROR. In the case of an exception, the **out** parameters are undefined.

```
procedure ALLOW_PENDING_JOINS
  ( CONNECTION : in CONNECTION_ID);
```

The ALLOW_PENDING_JOINS primitive accepts all pending opens which attempt to join the in-progress multicast connection indicated by the connection identifier, CONNECTION. If CONNECTION does not refer to a known connection, the primitive raises the exception CONNECTION_UNKNOWN_ERROR. If the connection is either not multicast or is in the GRACEFUL_CLOSE_SENT state, the primitive raises the exception

INVALID_OPERATION_ERROR. If the connection is in the TIMED_OUT state, the primitive raises the exception CONNECTION_TIMED_OUT_ERROR.

```
procedure CLOSE_CONNECTION
  ( CONNECTION             : in      CONNECTION_ID;
    MODE                   : in      TERMINATE_MODE;
    ACTIVITY_PARAMETERS    : in      ACTIVITY_BLOCK;
    INDEX                  :    out  LWCS.ACTIVITY_INDEX);
```

The CLOSE_CONNECTION primitive closes the connection identified by CONNECTION. The connection may be closed in one of two modes specified by the MODE parameter. If MODE is GRACEFUL, the connection is not closed until all outstanding sends on the connection are complete. If MODE is IMMEDIATE, the connection is closed immediately without regard to any outstanding sends on the connection. The ACTIVITY_PARAMETERS block provides the additional semantics for sending primitives described in section 2.2. The primitive also provides an activity index, INDEX. If CONNECTION is not known to exist, the primitive raises the exception CONNECTION_UNKNOWN_ERROR. If the connection is in the GRACEFUL_CLOSE_SENT state and MODE is GRACEFUL, the primitive raises the exception INVALID_OPERATION_ERROR. If the connection is in the TIMED_OUT state and MODE is GRACEFUL, the primitive raises the exception CONNECTION_TIMED_OUT_ERROR. In the case of an exception, the **out** parameter is undefined.

```
procedure CLOSE_CONNECTION_WITH_DATA
  ( CONNECTION             : in      CONNECTION_ID;
    ACTIVITY_PARAMETERS    : in      ACTIVITY_BLOCK;
    DATA                   : in      SYSTEM.ADDRESS;
    LENGTH                 : in      LWCS.DATA_BUFFER_SIZE;
    INDEX                  :    out  LWCS.ACTIVITY_INDEX);
```

The CLOSE_CONNECTION_WITH_DATA primitive inherits all the behavior of the CLOSE_CONNECTION primitive with MODE set to GRACEFUL and also includes the ability to transmit data as part of the connection shutdown procedure. The DATA parameter provides the base address of the contiguous data buffer to be transferred, and the LENGTH parameter specifies the length, in bytes, of the buffer. If any part of the data buffer is not at a valid memory address, the primitive raises the exception INVALID_PARAMETERS_ERROR. If the connection is in the GRACEFUL_CLOSE_SENT state or is a multicast receiving endpoint, the primitive raises the exception INVALID_OPERATION_ERROR. If the connection is in the TIMED_OUT state, the primitive raises the exception CONNECTION_TIMED_OUT_ERROR. In the case of an exception, the **out** parameter is undefined.

```
procedure CLOSE_ALL_CONNECTIONS
  ( MODE                     : in      TERMINATE_MODE;
    ACTIVITY_PARAMETERS      : in      ACTIVITY_BLOCK;
    INDEX                    :    out  LWCS.ACTIVITY_INDEX);
```

The CLOSE_ALL_CONNECTIONS primitive closes all existing connections of the application program. The MODE, ACTIVITY_PARAMETERS, and INDEX parameters have the same meaning as in the CLOSE_CONNECTION primitive. It is not an error to close all connections when none exist.

```
procedure UPDATE_CONNECTION_OPTIONS
  ( CONNECTION               : in      CONNECTION_ID;
    OPTIONS                  : in      LWPM.MESSAGE_OPTIONS);
```

The UPDATE_CONNECTION_OPTIONS primitive changes the message options which apply to each subsequent message sent on the connection identified by CONNECTION. The OPTIONS parameter provides the message options described in section 2.6. If CONNECTION does not exist, the primitive raises the exception CONNECTION_UNKNOWN_ERROR. If the connection is in the GRACEFUL_CLOSE_SENT state, the primitive raises the exception INVALID_OPERATION_ERROR. If the connection is in the TIMED_OUT state, the primitive raises the exception CONNECTION_TIMED_OUT_ERROR.

```
function GET_CONNECTION_STATE
  ( CONNECTION               : in      CONNECTION_ID)
    return CONNECTION_STATE;
```

The GET_CONNECTION_STATE primitive returns the state of the connection specified by CONNECTION. If CONNECTION is not known to exist, the primitive returns the state NONEXISTENT and does not raise an exception.

```
      procedure GET_REMOTE_ENDPOINT
        ( CONNECTION             : in      CONNECTION_ID;
          NETWORK                :    out  LWAM.PHYSICAL_ADDRESS;
          TRANSPORT              :    out  LWAM.PHYSICAL_ADDRESS);

   end LW_CONNECTION_MANAGEMENT;
```

The GET_REMOTE_ENDPOINT primitive returns the network and transport addresses, NETWORK and TRANSPORT, respectively, of the remote endpoint of the connection, CONNECTION. If CONNECTION is not known to exist, the primitive raises the exception CONNECTION_UNKNOWN_ERROR. If CONNECTION is in the ACCEPT_REQUEST_SENT state, the primitive raises the exception INVALID_OPERATION_ERROR.

2.4. LW_DATA_TRANSFER

The primitives in this package provide support for applications to send messages to one another. Messages may be sent or received along the byte stream of a connection or as a complete unit between specified endpoints.

```
with LW_ADDRESS_MANAGEMENT;
with LW_COMMUNICATIONS_SUPPORT;
with LW_CONNECTION_MANAGEMENT;
with LW_PROTOCOL_MANAGEMENT;
with SYSTEM;

package LW_DATA_TRANSFER is
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWCM renames LW_CONNECTION_MANAGEMENT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
```

```
procedure SEND_MESSAGE
  ( CONNECTION              : in      LWCM.CONNECTION_ID;
    ACTIVITY_PARAMETERS     : in      ACTIVITY_BLOCK;
    DATA                    : in      SYSTEM.ADDRESS;
    LENGTH                  : in      LWCS.DATA_BUFFER_SIZE;
    INDEX                   :    out  LWCS.ACTIVITY_INDEX);
```

The SEND_MESSAGE primitive sends a message over the connection specified by CONNECTION. The ACTIVITY_PARAMETERS block provides the additional semantics for sending primitives described in section 2.2. The DATA parameter provides the base address of the contiguous data buffer to be transferred, and the LENGTH parameter specifies the length, in bytes, of the buffer. The primitive also provides an activity index, INDEX. If any part of the data buffer is not at a valid memory address, the primitive raises the exception INVALID_PARAMETERS_ERROR. If CONNECTION does not exist, the primitive raises the exception CONNECTION_UNKNOWN_ERROR. If the connection is in the GRACEFUL_CLOSE_SENT state or is a multicast receiving endpoint, the primitive raises the exception INVALID_PARAMETERS_ERROR. If the connection is in the TIMED_OUT state, the primitive raises the exception CONNECTION_TIMED_OUT_ERROR. In the case of an exception, the **out** parameter is undefined.

```
procedure GET_MESSAGE
  ( CONNECTION              : in      LWCM.CONNECTION_ID;
    ACTIVITY_PARAMETERS     : in      ACTIVITY_BLOCK_POINTER;
    DATA                    : in      SYSTEM.ADDRESS;
    BUFFER_SIZE             : in      LWCS.DATA_BUFFER_SIZE;
    INDEX                   :    out  LWCS.ACTIVITY_INDEX);
```

The GET_MESSAGE primitive receives a message over the connection specified by CONNECTION. The ACTIVITY_PARAMETERS block provides the additional semantics for receiving primitives described in section 2.2. The DATA parameter provides the base address of the contiguous data buffer which is to hold received data, and the BUFFER_SIZE parameter specifies the size, in bytes, of the buffer. The primitive also provides an activity index, INDEX. If any part of the data buffer is not at a valid memory address and is used to receive data, the primitive raises the exception

INVALID_PARAMETERS_ERROR. If CONNECTION does not exist, the primitive raises the exception CONNECTION_UNKNOWN_ERROR. If the connection is in the GRACEFUL_CLOSE_SENT state or is a multicast sending endpoint, the primitive raises the exception INVALID_OPERATION_ERROR. If the connection is in the TIMED_OUT state, the primitive raises the exception CONNECTION_TIMED_OUT_ERROR. In the case of an exception, the **out** parameter is undefined.

```
procedure SEND_UNITDATA
  ( DESTINATION             : in      LWAM.ADDRESS_ID;
    SOURCE                  : in      LWAM.ADDRESS_ID;
    ACTIVITY_PARAMETERS     : in      ACTIVITY_BLOCK;
    DATA                    : in      SYSTEM.ADDRESS;
    LENGTH                  : in      LWCS.INITIAL_DATA_BUFFER_SIZE;
    OPTIONS                 : in      LWPM.UNITDATA_OPTIONS;
    INDEX                   :    out  LWCS.ACTIVITY_INDEX);
```

The SEND_UNITDATA primitive reliably sends a complete message to the remote endpoint referred to by the address identifier, DESTINATION, from the endpoint on the local host referred to by the address identifier, SOURCE. The ACTIVITY_PARAMETERS block provides the additional semantics for sending primitives described in section 2.2. The DATA parameter provides the base address of the contiguous data buffer to be transferred, and the LENGTH parameter specifies the length, in bytes, of the buffer. The OPTIONS parameter provides unitdata options as described in section 2.6. The primitive also provides an activity index, INDEX. If either DESTINATION or SOURCE are invalid, the primitive raises the exception INVALID_ADDRESS_ID_ERROR. If either the SOURCE or DESTINATION addresses are already in use, or the SOURCE address is not multicast and refers to an endpoint which is not on the local host, the primitive raises the exception INVALID_OPERATION_ERROR. If the underlying communications protocol is unable to allocate resources for the unitdata message, the primitive raises the exception UNABLE_TO_ALLOCATE_PROTOCOL_RESOURCES_ERROR. If the DESTINATION endpoint fails to accept the connection before a implementation-defined timeout expires, the primitive raises the exception COMMUNICATION_ATTEMPT_TIMED_OUT_ERROR. If any part of the data buffer is not at a valid memory address, the primitive raises the exception INVALID_PARAMETERS_ERROR. If the connection is in the GRACEFUL_CLOSE_SENT state, the primitive raises the exception INVALID_OPERATION_ERROR. If the connection is in the TIMED_OUT state, the primitive raises the exception CONNECTION_TIMED_OUT_ERROR. In the case of an exception, the **out** parameter is undefined.

```
    procedure GET_UNITDATA
      ( RECEIVER              : in      LWAM.ADDRESS_ID;
        ACTIVITY_PARAMETERS   : in      ACTIVITY_BLOCK_POINTER;
        DATA                  : in      SYSTEM.ADDRESS;
        BUFFER_SIZE           : in      LWCS.INITIAL_DATA_BUFFER_SIZE;
        INDEX                 :     out LWCS.ACTIVITY_INDEX);

end LW_DATA_TRANSFER;
```

The GET_UNITDATA primitive receives a complete message sent to the local address referred to by the address identifier, RECEIVER. The ACTIVITY_PARAMETERS block provides the additional semantics for receiving primitives described in section 2.2. The DATA parameter provides the base address of the contiguous data buffer which is to hold received data, and the BUFFER_SIZE parameter specifies the size, in bytes, of the buffer. The primitive also provides an activity index, INDEX. If RECEIVER is not valid, the primitive raises the exception INVALID_ADDRESS_ID_ERROR. If the RECEIVER address is already in use or is not multicast and refers to a remote host, the primitive raises the exception INVALID_OPERATION_ERROR. If the underlying communications protocol is unable to allocate resources necessary to receive the message, the primitive raises the exception UNABLE_TO_ALLOCATE_PROTOCOL_RESOURCES_ERROR. If any part of the data buffer is not at a valid memory address and is used to receive data, the primitive raises the exception INVALID_PARAMETERS_ERROR. In the case of an exception, the **out** parameter is undefined.

2.5. LW_ERROR_MANAGEMENT

The primitive in this package provides definitions of all errors and exceptions and provides support for an application to discover the error associated with an asynchronous primitive.

```
with LW_COMMUNICATIONS_SUPPORT;

package LW_ERROR_MANAGEMENT is
   package LWCS renames LW_COMMUNICATIONS_SUPPORT;

   type LW_ERROR is (ACTIVITY_UNKNOWN, COMMUNICATION_ATTEMPT_TIMED_OUT,
     CONNECTION_TIMED_OUT, CONNECTION_UNKNOWN, INVALID_ADDRESS_ID,
     INVALID_LOGICAL_NAME, INVALID_MAC_ADDRESS, INVALID_NETWORK_ADDRESS,
     INVALID_OPERATION, INVALID_PARAMETERS, INVALID_TRANSPORT_ADDRESS,
     LOGICAL_NAME_ALREADY_BOUND, LW_PROTOCOL_NOT_INITIALIZED, NONE,
     NONEXISTENT_BINDING, TRANSACTION_UNKNOWN,
     UNABLE_TO_ALLOCATE_PROTOCOL_RESOURCES,
     UNABLE_TO_INITIALIZE_LW_PROTOCOL,
     UNABLE_TO_TERMINATE_LW_PROTOCOL);

   ACTIVITY_UNKNOWN_ERROR, COMMUNICATION_ATTEMPT_TIMED_OUT_ERROR,
     CONNECTION_TIMED_OUT_ERROR, CONNECTION_UNKNOWN_ERROR,
     INVALID_ADDRESS_ID_ERROR, INVALID_LOGICAL_NAME_ERROR,
     INVALID_MAC_ADDRESS_ERROR, INVALID_NETWORK_ADDRESS_ERROR,
     INVALID_OPERATION_ERROR, INVALID_PARAMETERS_ERROR,
     INVALID_TRANSPORT_ADDRESS_ERROR, LOGICAL_NAME_ALREADY_BOUND_ERROR,
     LW_PROTOCOL_NOT_INITIALIZED_ERROR, NONEXISTENT_BINDING_ERROR,
     TRANSACTION_UNKNOWN_ERROR,
     UNABLE_TO_ALLOCATE_PROTOCOL_RESOURCES_ERROR,
     UNABLE_TO_INITIALIZE_LW_PROTOCOL_ERROR,
     UNABLE_TO_TERMINATE_LW_PROTOCOL_ERROR : exception;
```

The LW_ERROR type includes codes for all errors which may be returned by the GET_ERROR primitive below. Each error corresponds to an exception. These exceptions may be raised by one or more binding primitives in the event of an error.

```
   function GET_ERROR
     ( ID                      : in      LWCS.ACTIVITY_INDEX)
       return LW_ERROR;

end LW_ERROR_MANAGEMENT;
```

The `GET_ERROR` primitive returns the error, if any, associated with the asynchronous primitive referred to by the activity index, `ID`. If an error has occurred during the execution of the primitive after its call completed, it is returned and the activity referred to by ID becomes nonexistent. If there is no error associated with the primitive's execution, the primitive returns NONE and ID remains bound. If ID is not bound to an activity, this call raises the exception ACTIVITY_UNKNOWN_ERROR.

2.6. LW_PROTOCOL_MANAGEMENT

The primitives and data types declared in this package provide support for applications to initialize the underlying transport protocol, XTP, and expose or modify protocol parameters and defaults. The primitives are unique relative to those of other packages due to the fact that they are system activities. Hence, their scope is that of the host computer on which the application runs, not just that of the application program in which they are performed.

```
with LW_ADDRESS_MANAGEMENT;

package LW_PROTOCOL_MANAGEMENT is
   package LWAM renames LW_ADDRESS_MANAGEMENT;

   type ERROR_CONTROL_MODE is (AGGRESSIVE, NONE, NORMAL);
```

The ERROR_CONTROL_MODE type indicates the form of error control used by XTP. AGGRESSIVE corresponds to setting the XTP FASTNAK option, NONE corresponds to setting the NOERR option, and NORMAL corresponds to having the FASTNAK and NOERR options not set.

```
   type FLOW_CONTROL_MODE is (DISABLED, ENABLED, RESERVATION_MODE);
```

The FLOW_CONTROL_MODE type indicates the form of flow control used by XTP. DISABLED corresponds to setting the XTP NOFLOW option; ENABLED corresponds to not setting NOFLOW, and RESERVATION_MODE corresponds to setting the RES option.

```
   type ADDRESS_INITIALIZE_MODE is (ESTABLISH_ADDRESS,
     READ_PREDETERMINED_ADDRESS);
```

The ADDRESS_INITIALIZE_MODE type is used to specify dynamic or static host address assignment in the INITIALIZE_LW_PROTOCOL primitive discussed below.

```
subtype HOUR_DURATION is DURATION range 0.0 .. 3600.0;
```

HOUR_DURATION is used to limit the value of WAIT_TIMEOUT to one hour.

The CONNECTION_PARAMETERS, MESSAGE_OPTIONS, and UNITDATA_OPTIONS types are used in various primitives to allow an application access to parameters of the underlying transport service provider, the Xpress Transfer Protocol. Many of the parameters map directly to XTP protocol parameters listed in Figure 2-6 of reference [4]. Others map indirectly to protocol options found in the XTP header. For complete descriptions of the semantics of these protocol parameters and options, see references [4] and [5]. The correspondence of the Ada binding protocol parameters to XTP parameters and options is given by the table below.

Ada Binding Parameter	XTP Parameter/Option	Units
ALLOW_JOINS	*none*	*none*
BURST	burst	bytes
NO_CHECKSUM	NOCHECK	*none*
CONNECTION_TIME	CTIMEOUT	seconds
ERROR_CONTROL	FASTNAK, NOERR	*none*
FLOW_CONTROL	NOFLOW, RES	*none*
INACTIVITY_TIMEOUT	CTIMER	seconds
LIFETIME	ttl	seconds
MBUCKETS	Number of buckets	buckets
MULTICAST	MULTI	*none*
RATE	rate	bytes/second
WAIT_TIMEOUT	WTIMER	seconds

Table 4.1 Correspondence of Ada Binding and XTP Parameters/Options

Note that the `ALLOW_JOINS` parameter does not map to any XTP parameter or option; rather, it indicates that multicast receivers may join an in-progress multicast conversation on a given multicast connection at any time.

```
type CONNECTION_PARAMETERS is
  record
    BURST                    : POSITIVE;
    ERROR_CONTROL            : ERROR_CONTROL_TYPE;
    FLOW_CONTROL             : FLOW_CONTROL_TYPE;
    MBUCKETS                 : POSITIVE;
    MULTICAST                : BOOLEAN;
    RATE                     : POSITIVE;
  end record;
```

The `CONNECTION_PARAMETERS` type indicates parameters which apply to a connection over its entire lifetime. Such parameters may not be altered after the connection has been established.

```
type MESSAGE_OPTIONS is
  record
    ALLOW_JOINS              : BOOLEAN;
    NO_CHECKSUM              : BOOLEAN;
    CONNECTION_TIME          : DURATION;
    INACTIVITY_TIMEOUT       : DURATION;
    LIFETIME                 : DURATION;
    WAIT_TIMEOUT             : DURATION;
  end record;
```

The `MESSAGE_OPTIONS` type indicates options which apply to messages sent between communication endpoints. These options may be altered during the lifetime of a connection.

```
type UNITDATA_OPTIONS is
  record
    BURST                  : POSITIVE;
    NO_CHECKSUM            : BOOLEAN;
    CONNECTION_TIME        : DURATION;
    ERROR_CONTROL          : ERROR_CONTROL_TYPE;
    FLOW_CONTROL           : FLOW_CONTROL_TYPE;
    INACTIVITY_TIMEOUT     : DURATION;
    LIFETIME               : DURATION;
    MBUCKETS               : POSITIVE;
    MULTICAST              : BOOLEAN;
    RATE                   : POSITIVE;
    WAIT_TIMEOUT           : DURATION;
  end record;
```

The UNITDATA_OPTIONS type indicates options which apply to a unitdata transfer.

```
procedure INITIALIZE_LW_PROTOCOL
  ( MAC                    : in out  LWAM.PHYSICAL_ADDRESS;
    NETWORK                : in out  LWAM.PHYSICAL_ADDRESS;
    MODE                   : in      ADDRESS_INITIALIZE_MODE;
    CONNECTION_DEFAULTS    :    out  CONNECTION_PARAMETERS;
    MESSAGE_DEFAULTS       :    out  MESSAGE_OPTIONS;
    UNITDATA_DEFAULTS      :    out  UNITDATA_OPTIONS);
```

The INITIALIZE_LW_PROTOCOL primitive places the underlying transport service provider in its initial state, supplies it with the address of its host computer, and provides the application with the default protocol parameters. The meaning of the MAC and NETWORK parameters depends upon the value of ADDRESS_INITIALIZE_MODE. If this mode is set to ESTABLISH_ADDRESS, then the MAC and NETWORK parameters are used to set the media access control and network addresses which will identify the host. On the other hand, if the value of ADDRESS_INITIALIZE_MODE is set to READ_PREDETERMINED_ADDRESS, the MAC and network addresses of the host are set to predetermined values and these values are returned in the MAC and NETWORK parameters. The CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, and UNITDATA_DEFAULTS parameters are set to the default values configured into the transport protocol implementation. These defaults are returned by the service provider so that applications may use the settings recommended for use with the underlying protocol.

If the protocol has already been initialized, this call serves to re-initialize it. If MAC is not a valid MAC address, the call raises the exception INVALID_MAC_ADDRESS_ERROR. If NETWORK is not a valid network address, the call raises the exception INVALID_NETWORK_ADDRESS_ERROR. If the protocol fails to initialize itself within an implementation-defined time interval, the primitive raises the exception UNABLE_TO_INITIALIZE_LW_PROTOCOL_ERROR. In the case of an exception, the values of the **out** parameters are undefined. If an attempt to use the protocol is made before the protocol has been initialized, the primitive making the attempt shall raise the exception, LW_PROTOCOL_NOT_INITIALIZED_ERROR.

```
procedure TERMINATE_LW_PROTOCOL;
```

The TERMINATE_LW_PROTOCOL primitive places the underlying transport service provider in an inactive state. In such a state the protocol shall not perform any services until it is re-initialized. If the protocol is already terminated, the call has no effect and returns immediately. If the protocol fails to terminate itself within an implementation-defined time interval the primitive raises the exception UNABLE_TO_TERMINATE_LW_PROTOCOL_ERROR.

```
procedure UPDATE_PROTOCOL_PARAMETERS
  ( CONNECTION_DEFAULTS    : in     CONNECTION_PARAMETERS;
    MESSAGE_DEFAULTS       : in     MESSAGE_OPTIONS;
    UNITDATA_DEFAULTS      : in     UNITDATA_OPTIONS);
```

The UPDATE_PROTOCOL_PARAMETERS primitive provides the underlying transport service with new default protocol parameters. The CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, and UNITDATA_DEFAULTS parameters indicate the new default protocol parameter and option values that the implementation shall provide to inquiring applications. These defaults do not persist past re-initialization of the protocol.

```
    procedure RETURN_PROTOCOL_PARAMETERS
      ( MAC                     :    out  LWAM.PHYSICAL_ADDRESS;
        NETWORK                 :    out  LWAM.PHYSICAL_ADDRESS;
        CONNECTION_DEFAULTS     :    out  CONNECTION_PARAMETERS;
        MESSAGE_DEFAULTS        :    out  MESSAGE_OPTIONS;
        UNITDATA_DEFAULTS       :    out  UNITDATA_OPTIONS);

end LW_PROTOCOL_MANAGEMENT;
```

The `RETURN_PROTOCOL_PARAMETERS` primitive exposes the addresses identifying the host computer and current defaults for protocol parameters and options to be used with the underlying transport service. The `MAC` and `NETWORK` parameters are set to the current media access control and network addresses that identify the host. The `CONNECTION_DEFAULTS`, `MESSAGE_DEFAULTS`, and `UNITDATA_DEFAULTS` parameters are set to the default protocol parameter and option values recommended for use with the underlying transport protocol, XTP. If `MAC` is not a valid MAC address, the call raises the exception `INVALID_MAC_ADDRESS_ERROR`. If `NETWORK` is not a valid network address, the call raises the exception `INVALID_NETWORK_ADDRESS_ERROR`. In the case of an exception, the **out** parameters are undefined.

2.7. LW_TRANSACTION_SERVICES

The primitives in this package provide support for applications to initiate, accept, and respond to transactions.

```
with LW_ADDRESS_MANAGEMENT;
with LW_COMMUNICATIONS_SUPPORT;
with LW_DATA_TRANSFER;
with LW_PROTOCOL_MANAGEMENT;
with SYSTEM;

package LW_TRANSACTION_SERVICES is
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWDT renames LW_DATA_TRANSFER;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;

   type TRANSACTION_ID is private;
```

A TRANSACTION_ID is a handle on a transaction. A valid TRANSACTION_ID may only result from a successfully accepted transaction request and may only be used in sending a transaction response.

```
   procedure TRANSACTION_REQUEST
     ( DESTINATION           : in      LWAM.ADDRESS_ID;
       SOURCE                : in      LWAM.ADDRESS_ID;
       ACTIVITY_PARAMETERS   : in      ACTIVITY_BLOCK;
       REQUEST_DATA          : in      SYSTEM.ADDRESS;
       REQUEST_LENGTH        : in      LWCS.INITIAL_DATA_BUFFER_SIZE;
       RESPONSE_DATA         : in      SYSTEM.ADDRESS;
       BUFFER_SIZE           : in      LWCS.DATA_BUFFER_SIZE;
       OPTIONS               : in      LWPM.UNITDATA_OPTIONS;
       INDEX                 :     out LWCS.ACTIVITY_INDEX);
```

The TRANSACTION_REQUEST primitive initiates a transaction with the remote endpoint referred to by the address identifier, DESTINATION, from the endpoint on the local host referred to by the address identifier, SOURCE. The local endpoint transfers a request to the remote endpoint which later sends its response back to the local endpoint. The ACTIVITY_PARAMETERS block provides the dual semantics for both sending and receiving primitives described in section 2.2. The dual semantics are supported as follows. The ASYNCHRONOUS, PRIORITY and END_OF_MESSAGE parameters apply to the request. However, the BYTES_RECEIVED parameter applies to the response. The REQUEST_DATA parameter provides the base address of the contiguous data buffer to be transferred to the remote endpoint, and the REQUEST_LENGTH parameter specifies the length, in bytes, of the buffer. The RESPONSE_DATA parameter provides the base address of the contiguous data buffer which is to hold response data received from the remote endpoint, and the BUFFER_SIZE parameter specifies the size, in bytes, of the buffer.The OPTIONS parameter provides unitdata options as described in section 2.6. The primitive also provides an activity index, INDEX. If either DESTINATION or SOURCE are not valid, the primitive raises the exception INVALID_ADDRESS_ID_ERROR. If either the SOURCE or DESTINATION addresses are already in use, or the SOURCE address refers to an endpoint which is not on the local host, the primitive raises the exception INVALID_OPERATION_ERROR. If the underlying communications protocol is unable to allocate resources for the transaction, the primitive raises the exception UNABLE_TO_ALLOCATE_PROTOCOL_RESOURCES_ERROR. If the DESTINATION endpoint fails to accept the transaction request before a implementation-defined timeout expires, the primitive raises the exception COMMUNICATION_ATTEMPT_TIMED_OUT_ERROR. If any part of the data buffers lie outside the address space of the program and are used to hold transported data, the primitive raises the exception INVALID_PARAMETERS_ERROR. In the case of an exception, the **out** parameter and the response data are undefined.

```
procedure ACCEPT_TRANSACTION_REQUEST
  ( RECEIVER               : in      LWAM.ADDRESS_ID;
    ACTIVITY_PARAMETERS    : in      ACTIVITY_BLOCK_POINTER;
    REQUEST_DATA           : in      SYSTEM.ADDRESS;
    BUFFER_SIZE            : in      LWCS.INITIAL_DATA_BUFFER_SIZE;
    TRANSACTION            :    out  TRANSACTION_ID;
    INDEX                  :    out  LWCS.ACTIVITY_INDEX);
```

The ACCEPT_TRANSACTION primitive awaits a transaction request made to the local address referred to by the address identifier, RECEIVER. The ACTIVITY_PARAMETERS block provides the additional semantics for receiving primitives described in section 2.2. The primitive provides a unique transaction identifier, TRANSACTION, which provides a way to refer to the connection in the SEND_TRANSACTION_RESPONSE primitive described below. The REQUEST_DATA parameter provides the base address of the contiguous data buffer which is to hold the request data, and the BUFFER_SIZE parameter specifies the size, in bytes, of the buffer. The primitive also provides an activity index, INDEX. If RECEIVER is not a valid address identifier, the primitive raises the exception INVALID_ADDRESS_ID_ERROR. If the RECEIVER address is already in use or specifies a remote host, the primitive raises the exception INVALID_OPERATION_ERROR. If the underlying communications protocol is unable to allocate resources for the new connection, the primitive raises the exception UNABLE_TO_ALLOCATE_PROTOCOL_RESOURCES_ERROR. If any part of the data buffer is not at a valid memory address and is used to receive data, the primitive raises the exception INVALID_PARAMETERS_ERROR. In the case of an exception, the **out** parameters are undefined.

```
      procedure SEND_TRANSACTION_RESPONSE
        ( TRANSACTION             : in      TRANSACTION_ID;
          ACTIVITY_PARAMETERS     : in      ACTIVITY_BLOCK;
          RESPONSE_DATA           : in      SYSTEM.ADDRESS;
          RESPONSE_LENGTH         : in      LWCS.DATA_BUFFER_SIZE;
          INDEX                   :    out  LWCS.ACTIVITY_INDEX);

   end LW_TRANSACTION_SERVICES;
```

The SEND_TRANSACTION_RESPONSE primitive sends a response message to the endpoint which initiated the transaction identified by TRANSACTION. The ACTIVITY_PARAMETERS block provides the additional semantics for sending primitives described in section 2.2. The RESPONSE_DATA parameter provides the base address of the contiguous data buffer to be transferred as the transaction response, and the RESPONSE_LENGTH parameter specifies the length, in bytes, of the buffer. The primitive also provides an activity index, INDEX. If any part of the data buffer is not at a valid memory address, the primitive raises the exception INVALID_PARAMETERS_ERROR. If TRANSACTION is not known to exist, the primitive raises the exception TRANSACTION_UNKNOWN_ERROR. In the case of an exception, the **out** parameter is undefined.

References

[1] MIL_HDBK-818-1: *SAFENET Network Development Guidance*, United States Department of Defense, September, 1992.

[2] MIL_STD-2204: *Survivable Adaptable Fiber Optic Embedded Network (SAFENET)*, United States Department of Defense, September, 1992.

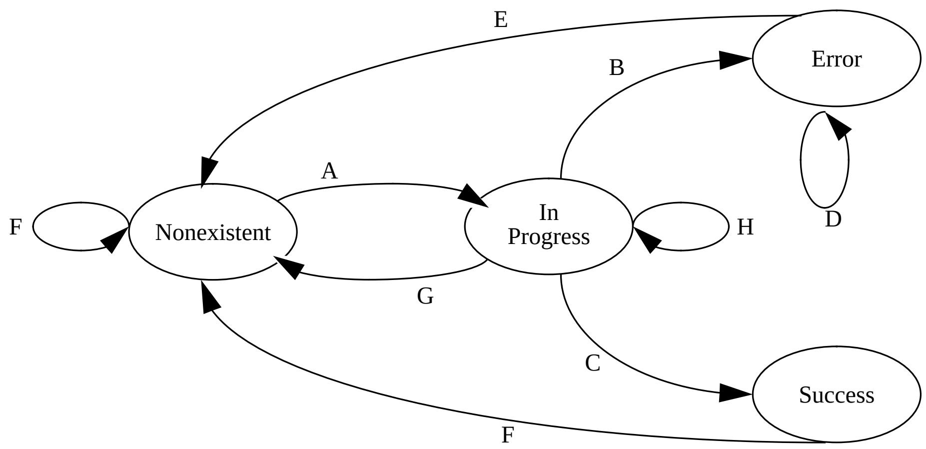
[3] Dempsey, Bert J., Fenton, John C., Michel, Jeffrey R., Waterman, Alexander S., “SAFENET SLA - Ada Binding,” University of Virginia, July 1992.

[4] *Xpress Transfer Protocol Definition: Revision 3.6*, Protocol Engines, Inc., Santa Barbara, California, January 1992.

[5] Strayer, W. Timothy, Dempsey, Bert J., and Weaver, Alfred C., *XTP: The Xpress Transfer Protocol*, Addison- Wesley, Reading, Massachusetts, 1992.

Appendix A: Activity State Model

The following diagram indicates the states which an activity may be in and the events which cause an activity to change state. The initial state of an activity is Nonexistent.



A: An asynchronous primitive call returns without an exception. The execution of the primitive is now referred to as an activity.

B: The activity encounters an error.

C: The activity completes successfully.

D: A call to `GET_ACTIVITY_STATE` is made for the activity.

E: A call to `GET_ERROR` or `WAIT_ON_ACTIVITY` is made for the activity.

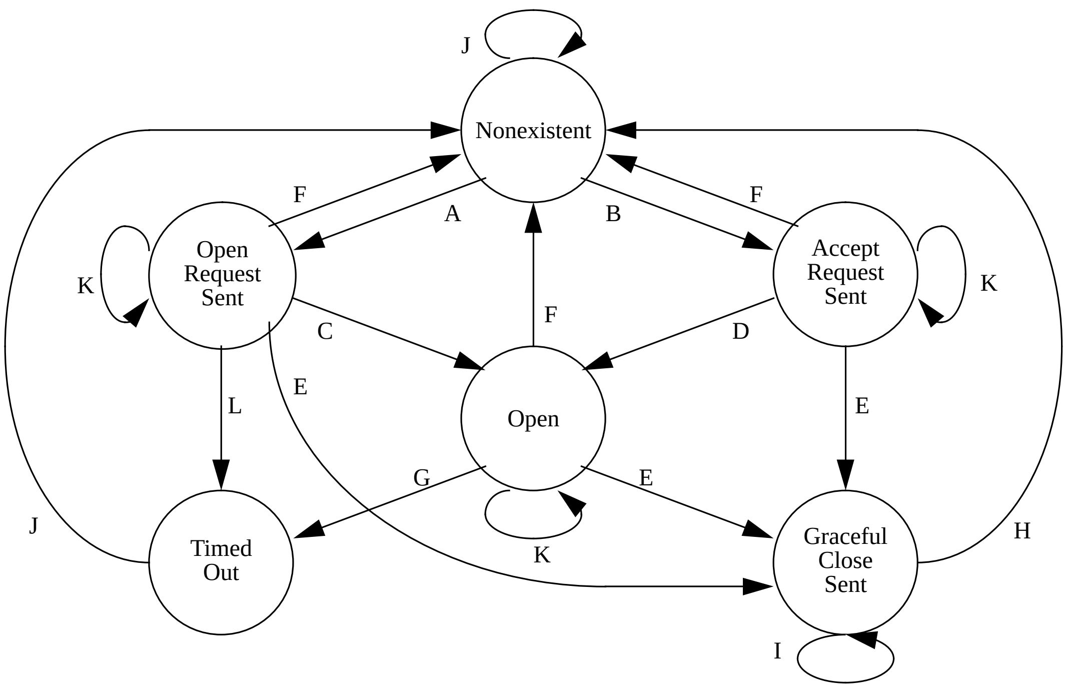
F: D or E.

G: A call to `WAIT_ON_ACTIVITY` is made for the activity.

H: A call to `GET_ACTIVITY_STATE` or `GET_ERROR` is made for the activity.

Appendix B: Connection State Model

The following diagram indicates the states which a connection may be in and the events which cause a connection to change state. The initial state of a connection is Nonexistent.



A: An opening primitive is awaiting a connection accept, but it has not yet arrived.

B: An accepting primitive is awaiting a connection request, but one has not yet arrived.

C: The connection request for this connection has been accepted.

D: A connection request has arrived for this connection.

E: A graceful close was issued for this connection.

F: An immediate close was issued for this connection.

G: The inactivity timer for this connection expired.

H: The graceful close issued for this connection has completed or F.

I: A `GET_CONNECTION_STATE` primitive was issued for this connection.

J: I or F.

K: A `SEND_MESSAGE`, `GET_MESSAGE`, `UPDATE_MESSAGE_OPTIONS`, `ALLOW_PENDING_JOINS`, or `GET_CONNECTION_STATE` primitive was issued for this connection.

L: The connection request was not accepted within an implementation-defined time interval.

The CLOSE_ALL_CONNECTIONS primitive is equivalent to a CLOSE_CONNECTION primitive applied to each open connection. For readability it is not referred to in the diagram. Note that there is no state transition on a call to the SEND_MESSAGE, GET_MESSAGE, ALLOW_PENDING_JOINS or UPDATE_MESSAGE_OPTIONS primitives or a graceful close for connections in states other than OPEN, OPEN_REQUEST_SENT, and ACCEPT_REQUEST_SENT. If such a call is made, an exception is raised. If the state is NONEXISTENT, the exception is CONNECTION_UNKNOWN_ERROR. If the state is GRACEFUL_CLOSE_SENT, the exception is INVALID_OPERATION_ERROR. If the state is TIMED_OUT, the exception is CONNECTION_TIMED_OUT_ERROR. The GET_REMOTE_ENDPOINT primitive has no effect upon the connection state model.