Writing Applications: Client and Server

This chapter describes how to implement and program client and server applications with EntireX Broker. It covers the following topics:

- Overview of Communication Models
- Basic Concepts of Client and Server
- API-TYPE and API-VERSION
- LOGON and LOGOFF
- USER-ID and TOKEN
- Control Block Fields and Verbs
- Implementation of Client and Server Components
- Blocked and Non-blocked Broker Calls
- Conversational and Non-conversational Mode
- Managing Conversation Contexts
- Delayed SEND Function
- Timeout Parameters
- Data Compression
- Error Handling
- Using Internationalization
- Using Send and Receive Buffers
- Tracing
- Transport Methods
- Variable-length Error Text
- Programmatically Turning on Command Logging

See also *Writing Applications: Attach Server* and *Writing Applications: Units of Work*.

Overview of Communication Models

There are two communication models in EntireX Broker: client and server and publish and subscribe.

- **Client and Server**
  This communication model is based on a logical connection between exactly two partners: a client and a server. It covers the communication requirements conversational and non-conversational, and synchronous and asynchronous.

- **Publish and Subscribe**
  This communication model is used if data is to be published to multiple recipients. It is an alternative to client and server and is implemented as an independent subsystem in EntireX Broker. See *Writing Applications: Publish and Subscribe*.

Basic Concepts of Client and Server

- Client-and-Server Application Components
- Conversationality
- Synchronicity

Client-and-Server Application Components

In the client-and-server communication model there are two partner application components: a requesting partner (the client) and the partner satisfying the request (the server). The client identifies the required service through the names of the `SERVER-CLASS`, `SERVER-NAME` and `SERVICE` with which the partner has registered.

EntireX Broker allows multiple server application components to register the same service in order to satisfy processing requirements. In conversational requests, the client and the server are bound to each other for the duration of the conversation. In addition, a server application component can satisfy more than one request type after registering several class, server and service names.

An application component is not restricted to a single role as either client or server; it can perform the role of both client and server. It can therefore make requests for processing while also satisfying requests from other partner application components.

Conversationality

The EntireX Broker allows both non-conversational and conversational communication in order to meet the different requirements of connections between distributed application components.

- **Non-conversational**
  In this communication type, each request comprises a single message from the client that requires at most one reply from a server. Since there is only one `SEND` / `RECEIVE` cycle per request, each request can be satisfied individually by any of a number of server replicas.

- **Conversational**
  In this communication type, the request contains a series of related messages, initiated by a client, which occur between client and server. Since there is a series of `SEND` / `RECEIVE` commands for

each request, the same replica of a server must process all related messages within a conversation.

Using EntireX Broker, an application may have more than one conversation active at the same time with the same partner or with different partners. Conversational and non-conversational modes can also be used simultaneously. The required mode of communication is always controlled by the application component that initiates the communication, that is, the client side.

Synchronicity

EntireX Broker makes possible both synchronous and asynchronous communication. EntireX Broker enables application components to combine synchronous and asynchronous communication as needed by the application. The terms synchronous and asynchronous correspond to the terms "blocked" and "non-blocked". See *Blocked and Non-blocked Broker Calls*.

- **Synchronous**
  The application component initiating the request waits for the processing to be completed by the partner application component before continuing. EntireX Broker provides the application with facilities to wait automatically for the partner application to complete processing and reply to the requesting application partner.

- **Asynchronous**
  The application component initiating the request does not wait for the processing to be completed and continues to execute without needing to receive a reply from the partner application. EntireX Broker provides the application with facilities to continue processing and obtain the partner’s reply at a later time, if needed.

API-TYPE and API-VERSION

Both the API-TYPE and the API-VERSION fields must always be provided.

Value	Bit Pattern	Description
1	(x'01')	The standard value for API-TYPE is 1 (x'01') and usable with all Broker stubs in all environments. **Note:** If any of the following conditions exist, you must install the Adabas CICS link module with the definition PARMTYP=ALL, using the ADAGSET macro. 1. If you are using NET transport with CICSETB stub with send or receive buffers greater than 32 KB. 2. If you are using NET transport with CICSETB stub and your application does not have a TWA.

Certain Broker functionality requires a minimum API-VERSION. Using publish and subscribe requires API-VERSION 8 or higher. For the highest available version of Broker, see API-VERSION. The send buffer and the receive buffer are passed as parameters to the EntireX Broker. Both buffers can occupy the same location.

See *Broker ACI Control Block Layout* for Assembler | C | COBOL | Natural | PL/I | RPG.

Both the API-TYPE and API-VERSION fields must be set correctly to ensure that Broker returns the correct value in ACI field ERROR-CODE. Otherwise, depending on your programming language and environment, a return code may not always be given.

See *Call Format* for Assembler | C | COBOL | Natural | PL/I | RPG.

LOGON and LOGOFF

The LOGON and LOGOFF Broker functions are optional when using the client-and-server programming model in your application. However, we recommend that the application issues LOGON and LOGOFF function calls for the following reasons:

- LOGOFF will notify the Broker to clean up in-memory resources held for your program, making them available to other users of the Broker.
- Without LOGOFF, the user's in-memory resources will time out in accordance with the Broker attributes CLIENT-NONACT and SERVER-NONACT. Depending on the values set by the administrator, this may not occur for some time.

Example for programming language Natural:

```
/* Logon to Broker/LOGON
MOVE #FCT-LOGON TO #ETBCB.#FUNCTION
/*
CALL 'BROKER' #ETBAPI #SEND-BUFF #RECV-BUFF #ERR-TXT
```

Logoff example for programming language Natural:

```
/* Logoff to Broker/LOGOFF
MOVE #FCT-LOGOFF TO #ETBCB.#FUNCTION
CALL 'BROKER' #ETBAPI #SEND-BUFF #RECV-BUFF #ERR-TXT
```

USER-ID and TOKEN

- Identifying the Caller
- Restarting after System Failure
- Managing the Security Token

Identifying the Caller

`USER-ID` identifies the caller and is required for all functions except `VERSION`. The `USER-ID` is combined with an internal ID or with the `TOKEN` field, if supplied, in order to guarantee uniqueness, for example where more than one application component is executing under a single `USER-ID`.

Brokers identify callers as follows:

- When the ACI field `TOKEN` is supplied:

  The ACI field `USER-ID`, together with the `TOKEN`, is used to identify the user. Using `TOKEN` allows the application to reconnect with a different process or thread without losing the existing conversation. When a new call is issued under the same `USER-ID` from a different location but with the same `TOKEN`, the caller is reconnected to the previous context.

  Note:
  The ability to reconnect to the previous context is vital if restart capabilities of applications are required. The combination of `USER-ID` and `TOKEN` must be unique to the Broker. It is not possible to have the same `USER-ID` and `TOKEN` combination duplicated.

- When the ACI field `TOKEN` is not supplied:

  The `USER-ID` is combined with an internally generated ID. It is possible to use the same `USER-ID` in different threads or processes. All threads and processes are distinct Broker users.

Restarting after System Failure

Warning:
`USER` and `TOKEN` must be specified by all publisher and subscriber applications where publication and subscription data is held in the persistent store.

The Broker provides a reconnection feature, using the `TOKEN` field in the ACI. If the application supplies a token along with `USER-ID`, the processing is automatically transferred when a request with the same user ID and token is received, either from the same process or from a different process or thread.

Specification of USER and TOKEN is necessary for reconnection with the correct user context after Broker has been stopped and restarted. This specification is also necessary to enable effective use of publish and subscribe, including recovery from system failures.

Managing the Security Token

If you are using EntireX Security, the application must maintain the content of the SECURITY-TOKEN field and not change this field on subsequent calls.

Control Block Fields and Verbs

- Basic Functionality of Broker API
- ACI Syntax
- Key ACI Field Names
- Key Verbs for FUNCTION Field

Basic Functionality of Broker API

This section describes the basic functionality of the Broker API. The following functions in the Broker API are fundamental to client-and-server processing. For full set of verbs relating to UOW processing, see *Control Block Fields and Verbs*.

- **DEREGISTER**
  The function `DEREGISTER` is used by a server to indicate its intention to terminate its role as a server for the named `SERVER-CLASS`, `SERVER-CLASS` and `SERVER-CLASS`. The server can terminate its role as server for all class, server and service names for which it is registered, using a single `DEREGISTER` command.

- **EOC**
  The function `EOC` is used by either partner to terminate one or more active conversations.

- **RECEIVE**
  The function `RECEIVE` is used by the server to obtain new requests from a client, and in the case of conversations, to obtain subsequent related messages from the same client. This function is also used by clients that issue asynchronous requests and wish to obtain the server's reply at a later time. The field `CONV-ID` defines the behavior of this function. `RECEIVE,CONV-ID=NEW` signals the server's readiness to obtain the next available new request, whereas the value `CONV-ID=`*nnn* indicates that the next message within an existing conversation is being requested by the server. The client uses `RECEIVE,CONV-ID=`*nnn* to obtain asynchronously a reply from the server for an existing conversation.

- **REGISTER**
  The function `REGISTER` is used by a component of an application to identify its intention to become a server and satisfy requests issued to the named `SERVER-CLASS`, `SERVER-CLASS` and `SERVER-CLASS`.

- **SEND**
  The function`SEND` is used by the client either to make a new request or to send subsequent related messages within a conversation. This function is also used by servers, after satisfying a request, or during the course of a conversation, to reply to the client. The field `CONV-ID` defines the behavior of this function. The client uses `SEND,CONV-ID=NEW` to initiate a new request and the value `CONV-ID=`*nnn* when sending subsequent related messages in a conversation. The server always uses `SEND,CONV-ID=`*nnn* when replying to a client, where *nnn* indicates the identity of the existing conversation. The same syntax is used for both conversational and non-conversational modes.

ACI Syntax

Function	Fields in EntireX Broker Control Block
DEREGISTER	API = 1 or higher , BROKER-ID = BROKER-ID , USER-ID = user_id [, TOKEN = token] , SERVER-CLASS = class_name | * , SERVER-NAME = server_name | * , SERVICE = service_name | * [, OPTION = QUIESCE | IMMED]
EOC	API = 2 or higher , BROKER-ID = BROKER-ID , USER-ID = user_id [, TOKEN = token] [, OPTION = CANCEL] , CONV-ID = conv_id | ANY [, SERVER-CLASS = class_name] [, SERVER-NAME = server_name] [, SERVICE = service_name]
RECEIVE	API = 1 or higher , BROKER-ID = BROKER-ID , USER-ID = user_id [, TOKEN = token] , WAIT = n | YES | NO , CONV-ID = conv_id | NEW | OLD | ANY , SERVER-CLASS = class_name | * , SERVER-NAME = server_name | * , SERVICE = service_name | *
REGISTER	API = 1 or higher , BROKER-ID = BROKER-ID , USER-ID = user_id [, TOKEN = token] , SERVER-CLASS = class_name , SERVER-NAME = server_name , SERVICE = service_name [, OPTION = ATTACH]
SEND	API = 1 or higher , BROKER-ID = BROKER-ID , USER-ID = user_id [, TOKEN = token] [, OPTION = DEFERRED] , WAIT = n | YES | NO , CONV-ID = conv_id | NEW , SERVER-CLASS = class_name , SERVER-NAME = server_name , SERVICE = service_name

Key ACI Field Names

The following table lists key ACI field names for implementing applications that use the client/server communication model. The other fields are available to identify partner programs, specify buffer lengths, convey error codes, etc.

See *Broker ACI Fields* for all fields.

ACI Field Name	Explanation
SERVER-CLASS	A client uses these fields to identify the service that it requires. A server uses this to offer a service.
CONV-ID	Identifier to obtain and specify the conversation. Also used to determine communication mode: conversational or non-conversational. See *Conversationality*.
FUNCTION	Function code for one of the verbs (see *Key Verbs for FUNCTION Field*).
OPTION	Indication of specific Broker behavior, depending on the function.
WAIT	Time value to specify blocking or non-blocking of the conversation. See *Blocked and Non-blocked Broker Calls*.

Key Verbs for FUNCTION Field

The following table lists the most important verbs for the FUNCTION field.

See *Broker ACI Functions* for a complete list of functions.

Verb	Description
REGISTER	Inform the EntireX Broker that a service is available.
RECEIVE	Retrieve request from partner.
SEND	Send reply to the partner.
EOC	Terminate one or more conversations.
DEREGISTER	Remove the availability of the service.

Implementation of Client and Server Components

This example implements a simple non-conversational server and the appropriate client. The server is able to receive a request from the client and send back a reply. See *Conversationality*.

The following EntireX Broker functions are used to implement the server component:

Function	Explanation
LOGON	Log on the application to EntireX Broker.
REGISTER	Inform EntireX Broker about the availability of a service.
RECEIVE	Retrieve request from partner.
SYNCPOINT	Commit the sending or acknowledgment receipts of a UOW and examine status.
SEND	Send reply to the partner.
DEREGISTER	Remove the availability of the service.
LOGOFF	Log off the application from EntireX Broker.

The program flow of the *client* component is:

```
LOGON USER-ID=user-id
SEND SERVER-CLASS=server-class,SERVER-NAME=server-name,SERVICE=service
LOGOFF USER-ID=user-id
```

The program flow of the *server* component is:

```
LOGON
REGISTER SERVER-CLASS=server-class,SERVER-NAME=server-name,SERVICE=service
repeat
     RECEIVE SERVER-CLASS=server-class,SERVER-NAME=server-name,SERVICE=service
     (individual request processing: reply to client for each message)
     SEND CONV-ID=n
end-repeat
DEREGISTER SERVER-CLASS=server-class,SERVER-NAME=server-name,SERVICE=service
LOGOFF
```

The example above illustrates the structure of a typical server program. It consists of a server registration and a loop with RECEIVE / SEND cycles. This RECEIVE / SEND loop is normally interrupted by shutdown messages from administration programs.

The appropriate client component needs three functions:

Function	Explanation
LOGON	Log on the application to EntireX Broker.
SEND	Send request to partner.
LOGOFF	Log off the application from EntireX Broker.

The service offered by the server above is used by issuing a SEND operation within the client component of the application.

Both server and client perform a LOGON as the first call and LOGOFF as the last call. This enables security checks and saves resources in EntireX Broker.

Blocked and Non-blocked Broker Calls

The application can use the EntireX Broker control block field `WAIT` to determine whether Broker will automatically generate a `WAIT` in order for the command to be received or satisfied by the partner application.

- Non-blocked Command: WAIT=NO
- Blocked Command: WAIT=YES or WAIT=n
- Examples: WAIT
- Examples: Programming Language Natural

Non-blocked Command: WAIT=NO

- **SEND**
  An application sends a message via Broker to a partner application. The caller does not wait for the partner application to `RECEIVE` the message or to process it. The application subsequently performs `RECEIVE` commands if it intends to retrieve messages from the partner. This technique is frequently used by server applications when replying to clients after satisfying their requests; it can also be used by client applications that do not want to wait for the request to be serviced, such as when using units of work (see *Writing Applications: Units of Work*).

- **RECEIVE**
  Allows an application to ask for a message to be returned from the partner application. If the partner application has not yet communicated any messages to Broker using the `SEND` command, an ACI response code is given to the application, indicating no messages are currently available either for the designated class/server/service or for the conversation (if an existing conversation was established). This technique can be used by both client and server application components, especially in a multithreading context, where more than one communication thread is being maintained, or when programming units of work (see *Writing Applications: Units of Work*).

Blocked Command: WAIT=YES or WAIT=n

- **SEND**
  An application sends a request via Broker to a partner application. The calling application is automatically put into a `WAIT` state until the partner application has performed a `RECEIVE` operation to obtain the request and then processes it before issuing a reply, using the `SEND` command. Unlike the case where `WAIT=NO`, an inherent `RECEIVE` is generated to return the partner's reply. This technique is used by client applications only.

- **RECEIVE**
  An application asks for a message to be returned from the partner application. The calling application is automatically put into a `WAIT` state until the partner application has provided the necessary message through issuing a `SEND` command. If no messages are available during the specified waiting time, an ACI response code is given to the application, indicating no messages were available for the designated class/server/service or for the conversation (if an existing conversation was established). This technique is frequently used by server applications when waiting for messages to arrive from clients; it can also be used by client applications if the `SEND` and `RECEIVE` commands are programmed separately.

Examples: WAIT

The EntireX Broker allows both server and client applications to specify a WAIT time with the SEND or RECEIVE function. WAIT is a field in the ACI control block (see *Broker ACI Fields*). If a WAIT time is specified, the application is suspended until a reply is received or the timeout value has elapsed. If a timeout occurs, the EntireX Broker returns an error code to the calling program. If no WAIT time is specified, the application continues processing and collects the reply later.

Server applications typically use the WAIT field with a RECEIVE function in order to wait for requests. WAIT is not typically used with server SEND functions, allowing the server to continue processing instead of waiting for a request. For example:

```
LOGON
REGISTER service
repeat
     RECEIVE,CONV-ID=NEW,WAIT=nS
     (individual processing)
     SEND,CONV-ID=n,WAIT=NO
end-repeat
DEREGISTER service
LOGOFF
```

Client applications use the WAIT field with a SEND function in non-conversational communication if they require a reply. Because the mode is non-conversational, no conversation ID is returned to the client. The client must therefore wait for the reply from the server.

```
LOGON
SEND,CONV-ID=NONE,WAIT=nS
LOGOFF
```

A RECEIVE function with no WAIT time can be used to check if requests or data/messages are available for processing. Control is returned to the caller even if no request or data/message is available to satisfy the caller’s operation. Appropriate error codes are returned when nothing is available.

```
LOGON
RECEIVE,CONV-ID=n,WAIT=NO
LOGOFF
```

The application can use the EntireX Broker control block field WAIT in the following ways to determine whether Broker will automatically generate a WAIT in order for the command to be received or satisfied by the partner application.

Examples: Programming Language Natural

- **Blocked Broker Calls**
  - Example 1: Single Request without Reply
  - Example 2: Single Request with Reply
- **Non-blocked Broker Calls**
  - Example 3: Long Running Service - Non-blocked Client

- Example 14: Single Requests without Reply - A Polling Server
- Example 15: Single Requests with Reply - A Polling Server

Note:
See Examples for EntireX Broker Tutorial.

Conversational and Non-conversational Mode

The mode of communication is always controlled by the component of the distributed application that initiates communication. In the client and server model, this is the client side. When starting a communication, the CONV-ID field of the ACI control block is used to signal the communication mode to the Broker as follows:

- **CONV-ID=NONE**
  Coded on the service-requesting side (client program), it denotes non-conversational mode. EntireX Broker assigns a unique conversation ID to the communication that the client does not need to know.

- **CONV-ID=NEW**
  Coded in the client program, it denotes conversational mode. The EntireX Broker assigns a unique conversation ID to the communication, which is retrieved by the server and client program. This conversation ID must be specified in subsequent calls by both sides to refer to this conversation, until the conversation is ended by either side.

The server always retrieves the unique conversation ID and uses it when sending back the reply to the client. If no reply is required in non-conversational mode, the server ignores the conversation ID.

Non-conversational Mode

When implementing a non-conversational communication, the CONV-ID field is used by the server as follows:

```
LOGON
REGISTER service
repeat
     RECEIVE,CONV-ID=NEW
     (individual processing)
     SEND,CONV-ID=n
end-repeat
DEREGISTER service
LOGOFF
```

The client's SEND function is supplemented as follows:

```
LOGON
SEND,CONV-ID=NONE
LOGOFF
```

Conversational Mode

When implementing conversational communication, the server uses the CONV-ID field as follows:

```
LOGON
REGISTER service
repeat
     RECEIVE,CONV-ID=NEW
    repeat
         (individual processing)
         SEND,CONV-ID=n
         RECEIVE,CONV-ID=n
    end-repeat until conversation ended
end-repeat
DEREGISTER service
LOGOFF
```

The conversation is ended when *Message Class 0003 - EntireX ACI - Conversation Ended* is received. See *Error Handling*.

The client's SEND function is supplemented as follows:

```
LOGON
SEND,CONV-ID=NEW
SEND,CONV-ID=n
SEND,CONV-ID=n
EOC,CONV-ID=n
LOGOFF
```

EOC Reason

The reason for an EOC might be of interest to the partner of the conversation. EntireX Broker enables you to define the CANCEL option for an EOC function to indicate an abortive end of conversation. You can also distinguish between a timeout and a regular EOC on the basis of the error number. The error class is always *Message Class 0003 - EntireX ACI - Conversation Ended*; the error number specifies the actual circumstances.

Examples: Programming Language Natural

- Non-conversational communication
  - Example 1: Single Request without Reply
  - Example 2: Single Request with Reply
- Conversational communication
  - Example 4: Transfer Messages from Server to Client
  - Example 5: Transfer Messages from Client to Server

Note:
See Examples for EntireX Broker Tutorial.

Managing Conversation Contexts

It is possible to program a server application to handle several clients simultaneously and thus many conversations in parallel. Such a server is also capable of providing several different services and this technique can be used to reduce the number of different server applications executing on your machine. This increases throughput without wasting resources on a new service replica. The following features make it easier to implement a server that supports multiple conversations:

- Conversation Status
- Conversation User Data
- Stored EOC

Conversation Status

The Broker ACI control block contains a field named `CONV-STAT`. This is filled by Broker after a `RECEIVE` command. The following values are possible:

Value	Description
NEW	This is a new conversation. If the server needs to allocate a user-specific area, for example, this can be done without a comparison being made against existing conversations.
NONE	This message is a conversationless message. It is probably not necessary to create a user context, since the next request of this user is completely independent of this one, which is a requirement of conversationless communication. The implementation of mixed servers (conversational and non-conversational) is easier if it is known whether a message is conversational or not.
OLD	The message belongs to an existing conversation. The server can refer to the conversation user data to find the partner context. See *Conversation User Data*.

Conversation User Data

Servers capable of serving multiple clients simultaneously are either stateless (servicing non-conversational requests) or they have to store conversation-related data for each user. This conversation-related context data is typically stored by the server application in a dynamic memory area. When a message is received, the user context related to that conversation must be located. This can be done by implementing a mapping structure in the application that can be indexed by the conversation ID, which returns the related context data.

Additionally, conversation-related contexts can be maintained by the Broker on behalf of the server application using the `USER-DATA` field in the ACI control block. Broker remembers information stored in the `USER-DATA` field when executing the `SEND` command. This data is returned to the application on subsequent `RECEIVE` commands executed within the same conversation. Therefore, your application is able to store information in `USER-DATA` when executing `SEND` commands and retrieve it on `RECEIVE` commands. The data in `USER-DATA` is considered binary and is untouched by the Broker.

Note:
The `USER-DATA` is never transmitted from client to server or vice versa. Both sides of a conversation can store different `USER-DATA`, and both sides always receive their own data.

This USER-DATA helps with context areas as follows. A server application encounters a new conversation with the CONV-STAT API field. The user area is created and, typically, a first application confirmation is sent back to the client. Along with this SEND function, the server specifies the pointer to the user context - or the index into a context array, or whatever is available - into the USER-DATA. Whenever another request/message comes from that client via this conversation, this pointer/index is returned to the application, and the server has the context of the client application immediately, without having to scan a list of known conversations. Example:

```
* example of State-ful server program which utilizes
* USER-DATA to maintain application specific context
* information between successive messages within
* conversations with clients.

REGISTER #SERVER-CLASS #SERVER-NAME #SERVICE

  DO FOREVER
    RECEIVE #CONV-ID=ANY
    DECIDE ON FIRST VALUE #ERROR-CODE
      /* ============================
      /* NICE RETURN CODE
      VALUE '0'
        DECIDE ON FIRST VALUE #CONV-ID
          /* ========================
          /* NEW CONVERSATION
          VALUE 'NEW'
            #REQUEST-IN = #RECEIVE-BUFFER
            ... PROCESS NEW REQUEST FROM CLIENT AND
                REPLY TO CLIENT ASKING BROKER TO REMEMBER
                ACCOUNT NUMBER SO CLIENT DOESN'T HAVE TO
                TRANSMIT THIS WITH EVERY MESSAGE
            #ACCOUNT-NR = REQUEST-IN.ACCOUNT-NR
            SEND #CONV-ID #SEND-DATA #USER-DATA
          /* ========================
          /* EXISTING CONVERSATION
          NONE VALUE
            /* NEXT MESSAGE IN CONVERSATION RECEIVED
            /* AND ACCOUNT NUMBER REMEMBERED BY BROKER
            #ACCOUNT-NR = #USER-DATA
            #REQUEST-IN = #RECEIVE-BUFFER
            ... DO SOME PROCESSING BASED ON REQUEST AND
                ACCOUNT NUMBER REMEMBERED BY BROKER FOR
                THIS CONVERSATION CONTEXT
            ... REPLY TO CLIENT AS APPROPRIATE AND
                END CONVERSATION SOONER OR LATER
            SEND #CONV-ID #SEND-DATA #USER-DATA
        END-DECIDE
      VALUE '00740074'                /* RECEIVE TIME-OUT
        ESCAPE BOTTOM
      NONE VALUE                      /* REAL BROKER ERROR
        ... DEAL WITH A REAL BROKER ERROR
    END-DECIDE
  DOEND /* END FOREVER LOOP

DEREGISTER
```

Stored EOC

Servers that handle multiple conversations in parallel normally have to maintain a user context related to every conversation as described above. However, this context is typically allocated dynamically, and is therefore released after the conversation has ended. Not knowing when a particular conversation has finished would result in orphan contexts. To avoid this, the Broker offers the `NOTIFY-EOC` option, which is a service-specific attribute defined in the *Broker Attributes*.

This means that the `EOC` notification, even for timed-out conversations, is kept until the server receives it. This is useful for servers serving multiple conversations, since they are always informed about the end of a particular conversation and can therefore release all internal resources of a particular user context.

Specification of NOTIFY-EOC=YES can consume substantial system resources; as a result, a shortage of conversations for a service may occur. To avoid this shortage, a server must issue RECEIVE requests not restricted to any conversation, which gives the Broker the chance to report timed-out conversations. This does not of course mean that only RECEIVE functions with CONVERSATION-ID=ANY are valid, but from time to time such an unrestricted RECEIVE function should be issued.

Delayed SEND Function

To allow maximum flexibility in communication, the EntireX Broker provides a simple means of delaying the delivery of messages: allowing delivery of related messages in one logical block. If, for some reason, the messages that belong to a block cannot all be sent, all the messages in the logical block can optionally be deleted.

The mechanisms by which the EntireX Broker does this are the HOLD option on the SEND function and the UNDO function. Messages sent with HOLD status are not delivered until a message without the HOLD option is sent on the same conversation.

Example

This example illustrates the logical program flow of a client program that sends several messages on the same conversation, making delivery of the messages dependent on some condition. If the logical block of messages cannot be delivered (triggering an error condition), all messages in the logical block already sent can be deleted:

```
SEND,CONV-ID=NEW,OPTION=HOLD
....                                /* individual processing
SEND,CONV-ID=n,WAIT=NO,OPTION=HOLD
....                                /* individual processing
SEND,CONV-ID=n,WAIT=NO,OPTION=HOLD
....                                /* individual processing
if <error> then                     /* error condition
    UNDO,CONV-ID=n,OPTION=HOLD
else
    SEND,CONV-ID=n,WAIT=NO
end-if
....                                /* individual processing
EOC
```

Example: Programming Language Natural

- Example 7: Send Messages with HOLD - Delayed Delivery

Timeout Parameters

- Timeout Behavior
- Types of Non-activity Time
- Recommendations
- Unit of Work Lifetime

- Unit of Work Status Lifetime

Timeout Behavior

EntireX Broker provides a number of timeout mechanisms that allow you to control WAIT times flexibly, optimize resource usage, and configure efficient communication.

- The CLIENT-NONACT, SERVER-NONACT and CONV-NONACT attributes are non-activity timeout parameters that can be specified independently of each other to govern the three elements involved in a conversation: the requesting client, the registered server, and the conversation that will exist between them.
- The WAIT field in the Broker ACI control block allows you to place the sending or receiving program in a WAIT state for a specified time to allow data or a reply to be received before control is passed to the calling program. Placing the program into a WAIT state during a Broker command is referred to as issuing a blocked command. A non-blocked command is executed if WAIT=NO is specified. See *Blocked and Non-blocked Broker Calls*.

There is interplay between the WAIT values of your SEND and RECEIVE calls and the settings of the non-activity parameters in the Broker attribute file. See the WAIT field.

Types of Non-activity Time

There is interplay between the non-activity times specified in the attribute file for the attributes

- PUBLISHER-NONACT
- SUBSCRIBER-NONACT
- CLIENT-NONACT and
- SERVER-NONACT

where an application component performs more than one of these roles. In this case, the maximum non-activity time associated with the user will take precedence. This fact must be considered where an application component implements both publish and subscribe and client and server.

Recommendations

The following recommendations apply to developing client and server applications:

- Make the Broker WAIT time used for blocked SEND / RECEIVE calls in the application (both servers and clients) adjustable. This means that WAIT values must be read as a startup parameter from a user-supplied INI or CFG file, or any other parameter data set or set of environment variables, depending on the platform in use.
- On the client side, avoid high values for the WAIT time, which may lead to communication problems.
- When the WAIT time is lower than CONV-NONACT attribute, the caller will receive 00740074 error messages. Since the lifetime of the conversation exceeds the WAIT time specified for the command, the application can retry with the Broker function RECEIVE, and option LAST is possible.

- When the WAIT time is higher than CONV-NONACT attribute, the caller will receive 00030003 error messages. Since the lifetime of the conversation is less than the WAIT time specified for the command, it is not possible for the application to retry because any messages relating to the current conversation have already been cleaned up.

See also *Timeout Considerations for EntireX Broker*.

Unit of Work Lifetime

The UWTIME parameter in the *Broker Attributes* specifies the lifetime for a persistent UOW. The UOW exists until it has been successfully processed or until it is explicitly cancelled or backed out. If a UOW times out before being processed, or before any other explicit action is taken, its status changes to TIMEOUT. The status may or may not be retained in the persistent store, depending on the value of UOW status lifetime as described below. The default UOW lifetime for the Broker is defined by the UWTIME attribute. It can be overridden by the application in the UWTIME field of the ACI control block.

The UOW lifetime for the units of work is calculated only while Broker is executing.

Unit of Work Status Lifetime

This can be specified through either of the following two exclusive attribute settings. The default value zero implies the UOW status lifetime is zero, which means the status of the UOWSTATUS is not retained after one of the following events occurs: UOW is processed; UOW times out; UOW is backed out; UOW is cancelled. Status lifetime can be specified through either of the following two parameters in the *Broker Attributes*:

- UWSTATP (ACI_VERSION 3 or above)

  This attribute contains a multiplier used to compute the lifetime of the status of a UOW. See *Writing Applications: Units of Work*. The UWSTATP value is multiplied by the UWTIME value (the lifetime of the associated UOW) to determine how much additional time the UOW status is retained in the persistent store. The lifetime is calculated to start when any of the above events occurs and ends when the lifetime value expires. It can be overridden by the application in the UOW-STATUS-PERSIST field in the ACI control block.

- UWSTAT-LIFETIME (ACI_VERSION 8 or above)

  This attribute specifies the value to be added to the UWTIME (lifetime of the associated UOWSTATUS) to compute the length of time the UOW status is persisted. The UOW status lifetime begins at the time at which the associated UOW enters any of the following statuses: PROCESSED, TIMEOUT, BACKEDOUT, CANCELLED, DISCARDED. Specifying unit of work status lifetime in this way excludes specifying it as a multiplier value through the attribute UWSTATP.

The status lifetime for the unit of work is calculated only while Broker is executing.

Note:
The values described here as UWSTATP and UWSTAT-LIFETIME can also be assigned as global Broker attributes or as a per-service attribute. However, the value specified by the application in the ACI control block overrides the Broker (or service) attributes. See *Broker ACI Fields*.

Data Compression

Data compression within EntireX Broker allows you to exchange smaller packet sizes between senders and receivers. This helps to reduce response time during transmissions as well as improve the overall network throughput, especially with low bandwidth connections.

Compression is performed only on the buffers used to send and receive data. The application has the option of setting the level of compression/decompression for data transmission. The compression level can be set to achieve either no compression or a range of compression/decompression. See *Data Compression in EntireX Broker*. Application components can set compression individually to Broker.

zlib is a general-purpose software implementing data compression across a variety of platforms. The functions used within EntireX Broker represent a subset of those available within the zlib software. The compression algorithms are implemented through the open source software zlib. It may occur that the data buffer does not compress during a data transmission; if it does not compress, a logged warning message will appear in 00200450 and in the stub.

Technique

The Broker ACI control block contains a field that is used to set the compression level. This field determines for any send/receive transmission whether the data buffer will be compressed/decompressed. See ACI control block field `COMPRESSLEVEL`.

Error Handling

After every broker operation, the application must check the `ERROR-CODE`. It consists of a combination of

- error class (first four digits) and
- error number (last four digits)

While the error number describes the exact situation, the error class often determines how the program will proceed after returning from the EntireX Broker operation. From the programmer’s point of view, therefore, the error class may be more important than the particular error number.

For more information, see *Error Messages and Codes*.

Programming Techniques

We recommend trapping the error classes in a "case" statement, for example, a `DECIDE` in Natural or a switch statement in C.

All error classes - for example user and configuration errors - leading to the same action (that is, reporting or logging the situation and aborting issuing broker calls), can be handled together in the `NONE VALUE` or default case.

Example for C Progamming Language

```
int   i, iErrorCode, iErrorClass, iErrorNumber, ret_val;
char  szErrorTextBuffer[S_TXT + 1];.....

/* prepare error code field and error text buffer */
memset(pETBCB->error_code,'0',sizeof(pETBCB->error_code));
memset(szErrorTextBuffer,'\0',sizeof(szErrorTextBuffer));

/* call the broker */
ret_val = broker(pETBCB,pSendBuffer,pReceiveBuffer,szErrorTextBuffer);

/* evaluate error class from error code field */
iErrorClass = 0;
for(i = 0; i < 4; ++i)
{
    iErrorClass *= 10;
    iErrorClass += pETBCB->error_code[ i ] - '0';
}

if (iErrorClass == 0 && ret_val != 0)
{
    printf("Wrong API_TYPE and/or API_VERSION\n");
}
else
{
    /* evaluate error number from error code field */
    iErrorNumber = 0;
    for(i = 4; i < 8; ++i)
    {
        iErrorNumber *= 10;
        iErrorNumber += pETBCB->error_code[ i ] - '0';
    }

    /* evaluate error code as integer value */
    iErrorCode = (iErrorClass * 10000) + iErrorNumber;

    /* handle error */
    switch (iErrorClass)
    {
        case 0: /* Successful Response */
            ....
            break;

        case 2: /* User does not exist */
            ....
            break;

        case 3: /* Conversation ended */
            ....
            break;

        case 7: /* Service not registered */
            ....
            break;

        case 74: /* Wait Timeout occurred */
            ....
            break;

        ....
```

```
        default:
            printf("EntireX Broker Error occurred.\n");
            printf("%8.8u %s",iErrorCode,szErrorTextBuffer);
            break;
    }
}
```

Using Internationalization

It is assumed that you have read the document *Internationalization with EntireX* and are familiar with the various internationalization approaches described there.

This section covers the following topics:

- General Information
- Providing Locale Strings
- Using the ENVIRONMENT Field with the Translation User Exit

General Information

The broker stub does not convert your application data before it is sent to the broker. The application's data is shipped as given.

For the internationalization approaches ICU conversion and SAGTRPC user exit, valid locale strings are required for conversion to behave correctly.

Providing Locale Strings

Under the Windows operating system:

- The broker stub assumes by default that the data is given in the encoding of the Windows ANSI codepage configured for your system. If you are using at least `API-VERSION` 8 and communicating with a broker version 7.2.*n* or above, a codepage identifier of this Windows ANSI codepage is also automatically transferred as part of the locale string to tell the broker how the data is encoded.
- If you want to adapt the Windows ANSI codepage, see the Regional Settings in the Windows Control Panel and your Windows documentation.

Under all other operating systems:

- The broker stub does not automatically send a codepage identifier to the broker as part of the locale string.
- The broker stub assumes the broker's locale string defaults match. If they do not match, provide the codepage explicitly. See *Broker's Locale String Defaults*.

With the ACI control block field `LOCALE-STRING`:

- You can override or provide a codepage in the locale string sent to the broker. If a codepage is provided, it must follow the rules described under *Locale String Mapping*.
- You can force a locale string to be sent if communicating with broker version 7.1.x and below. Under Windows you can use the abstract codepage name. See *Using the Abstract Codepage Name LOCAL*.
- API version 4 or above is required to override the locale string.

The encoding in which your application gives the data to the broker stub and the locale string

- must always match, i.e. the codepage derived after the broker's built-in locale string mapping process must be the same as the encoding of the data provided. See *Broker's Built-in Locale String Mapping*.
- must be a codepage supported by the broker, depending on the internationalization approach;

otherwise, unpredictable results will occur.

Example for Assembler

```
MVC   S$LOCALE,=C'ECS037'                MOVE CP
....
```

Examples for C

1. Using a specific codepage

   ```
   /* prepare the locale-string with a codepage */
   memset (pETBCB->locale_string,' ',sizeof(pETBCB->locale_string));
   strncpy(pETBCB->locale_string,"ECS0819",sizeof(pETBCB->locale_string));
   ....
   ```

2. Using the platform's default codepage (Windows only)

   ```
   /* prepare the locale-string with a codepage */
   memset (pETBCB->locale_string,' ',sizeof(pETBCB->locale_string));
   strncpy(pETBCB->locale_string,
   ETB_CODEPAGE_USE_PLATFORM_DEFAULT,sizeof(pETBCB->locale_string));
   ....
   ```

Example for COBOL

```
MOVE 'ECS037' TO LOCALE-STRING.
....
```

Examples for Natural

```
MOVE 'ECS037' TO #SDPA-API.#LOCALE_STRING.
....
```

Using the ENVIRONMENT Field with the Translation User Exit

Using the internationalization approach *translation user exit*, an ACI programmer can provide additional information to their translation exit through the `ENVIRONMENT` field, allowing flexible translation behavior in accordance with application requirements. The field cannot be used for any other internationalization approaches and must be empty if a method other than translation user exit is used. See *Translation User Exit*.

Example

Assume a broker service or topic has a user-written translation routine called ABCTRAN, which is capable of performing several types of data conversion, for example EBCDIC-ASCII translation, byte swapping, and mixed data types. The user translation routine may need to know the data formats used by both partners. The `ENVIRONMENT` field can be used to pass this information from the application to the translation routine in Broker kernel.

Technique

```
MOVE 'MYCODEPAGE' TO #ETBCB.#ENVIRONMENT
...
CALL 'BROKER' #ETBAPI #SEND-BUFF #RECV-BUFF #ERR-TXT
```

Using Send and Receive Buffers

Introduction

The send buffer and the receive buffer are passed as parameters to the EntireX Broker. Both buffers can occupy the same location. See *Call Format* for Assembler | C | COBOL | Natural | PL/I | RPG.

The length of the data to be sent is given in the ACI field `SEND-LENGTH`. If the `SEND-LENGTH` is greater than the send buffer during data transmission, you could accidentally send the data that is physically located in memory behind your send buffer to the designated Broker.

The `RECEIVE-LENGTH` is required with the `RECEIVE` and `RECEIVE_PUBLICATION` functions and with `SEND` functions waiting for a reply. The length of the receive buffer is specified in the ACI field `RECEIVE-LENGTH`. If the `RECEIVE-LENGTH` is greater than the receive buffer during data reception, you can overwrite the data physically located behind the receive buffer being used.

If the data to be returned is less than `RECEIVE-LENGTH`, the rest of the receive buffer remains unchanged and is not padded with trailing blanks or other characters. The ACI field `RETURN-LENGTH` contains the length of the data actually returned. The `RECEIVE-LENGTH` field is not changed upon return.

Note:
With Adabas version 8, the maximum size of message data is no longer limited to approximately 32 KB. If Adabas version 8 is not used, these same limits still apply under z/OS.

Error Cases

Conversion and translation of data can increase the amount of data and thus require a buffer of a larger size than provided. It may also be impossible to determine the size required in advance. EntireX provides a feature to reread the data in such cases:

Using API version 2 and above, if the amount of data to be returned is greater than the `RECEIVE-LENGTH`, the exact length needed is given in the ACI field `RETURN-LENGTH` together with an error code, depending on the internationalization approach. See *Internationalization with EntireX*. Note the following:

For translation and translation user exit:

- The error code is 00200094.
- The data up to the length of the receive buffer is translated. The rest is truncated.

for ICU conversion and SAGTRPC user exit:

- The error code is 00200377.
- No data is returned in the receive buffer.

To obtain the entire message, increase the size of the receive buffer and issue an additional Broker ACI function `RECEIVE` or `RECEIVE_PUBLICATION` with the option "LAST".

Using API version 5 and above, it is also possible for a client to reread a truncated message in non-conversational mode, by issuing an additional Broker ACI function `RECEIVE` or `RECEIVE_PUBLICATION` with the option "LAST" as well as the `CONV-ID` returned from the ACI control block. No `EOC` is needed after `RECEIVE`.

Transport Methods

The maximum length possible for send and receive buffers is affected by the transport method used.

Transport Method	Maximum Receive / Send Buffer Size	If using this transport method, ...
TCP/IP	2,147,482,111 B	• the maximum send and receive buffer size is approximately 2,147,482,111 bytes.
Entire Net-Work	30,545 B	• the send and receive buffer sizes are affected by the setting of the Net-Work parameter `IUBL` for all involved platforms (see the Net-Work documentation for more information); • the send and receive buffer sizes are affected by the Adabas SVC/Entire Net-Work-specific attribute `IUBL` for Broker running under z/OS; • the maximum send and receive buffer size is around 30,545 bytes. **Note:** Under z/OS with Adabas version 8, the value for NET is the same as for TCP and SSL.
SSL	2,147,482,111 B	• the maximum send and receive buffer size is approximately 2,147,482,111 bytes.

Tracing

Trace information showing the commands help the application programmer debug applications and solve problems. Tracing can be obtained for the application (stub trace) and for the Broker kernel (kernel trace). The stub trace shows the Broker functions issued by your application, whereas the Broker kernel trace will contain all Broker functions issued by all applications using the Broker.

Setting the Broker attribute `TRACE-LEVEL=1` provides traces containing just the Broker functions processed by the Broker kernel without additional diagnostics. It is only necessary to set the trace value higher when generating traces for Software AG support.

Stub Trace

Tracing is available for all stubs on UNIX and Windows. For the stubs for which tracing is available on z/OS, see table under *Administering Broker Stubs*.

To set the stub trace, see *Tracing for Broker Stubs* under z/OS | UNIX | Windows | z/VSE.

Kernel Trace

Tracing is available for Broker on all platforms. For z/OS, see *Administering Broker Stubs*.

To set the kernel trace, see *Tracing webMethods EntireX* under UNIX | Windows | BS2000/OSD | z/VSE.

Transport Methods

Overview of Supported Transports

This table gives an overview of the transport methods supported by EntireX Broker stubs.

Operating System	Environment	Module	Transport to Broker			
			TCP	SSL	NET (1)	HTTP(S) (6)
z/OS (2)	Batch, TSO, IMS (BMP)	BROKER	x	x	x	
	Com-plete	COMETB	x	(3)	x	
	CICS	CICSETB	x	(3)	x	
	IMS (MPP)	MPPETB	x	x	x	
	IDMS/DC (4)	IDMSETB	x	(3)		
	Natural	NATETB23	x	x	x	
	UNIX System Services	*Java ACI*	x	x		x
UNIX		broker.so	x	x		
		Java ACI	x	x		x
Windows		broker.dll (5)	x	x		
		Java ACI	x	x		x
BS2000/OSD	Batch, Dialog (formerly TIAM)	BROKER	x	x	x	
z/VSE	Batch	BKIMB	x		x	
	CICS	BKIMC	x		x	
z/VM		BKIMBCMS	x		x	
IBM i		EXA	x			
OpenVMS		BROKER	x	x		

Notes:

1. NET is available for transport to a broker running under mainframe platforms only; not to a broker running under UNIX or Windows.
2. Under z/OS you can use IBM's Application Transparent Transport Layer Security (AT-TLS) as an alternative to direct SSL support inside the broker stub. Refer to the IBM documentation for more information.
3. Use AT-TLS. See Note 2.
4. Tracing and transport timeout are not supported in this environment.
5. Stub broker32.dll is supported for reasons of backward compatibility. The functionality is identical to broker.dll.
6. Via Broker HTTP(S) Agent; see *Settting up and Administering the Broker HTTP(S) Agent* under UNIX | Windows.

See also:

- *Transport Methods for Broker Stubs* under z/OS | UNIX | Windows | BS2000/OSD | z/VSE | z/VM
- *Setting Transport Methods* under *Writing Advanced Applications - EntireX Java ACI*

TCP/IP

TCP is not available for all Broker stubs and all environments (see table above).

See *Using TCP/IP as Transport Method for the Broker Stub* in *Transport Methods for Broker Stubs* under z/OS | UNIX | Windows | BS2000/OSD | z/VSE | z/VM, which describes how to set up TCP transport.

Application programs using TCP/IP as the transport specify the target Broker ID in terms of a host name (or IP address) together with the port number on which the Broker TCP/IP communications driver is listening. Example: An application communicating through TCP/IP would specify on each command the Broker ID

```
IBM1:3932:TCP
```

where the host on which the Broker kernel executes is known to TCP as IBM1 and is listening on port 3932.

Entire Net-Work

Communication through Entire Net-Work is available for all Broker stubs when communicating with a Broker kernel on z/OS through Entire Net-Work. Applications can also utilize Entire Net-Work communication to obtain local interprocess communication with a z/OS Broker kernel running on the same machine as the application. This can provide a considerable performance benefit. Local interprocess communication is achieved through the Adabas SVC mechanism.

Application programs using Entire Net-Work as the transport specify the target Broker ID in terms of the target Entire Net-Work ID of the Broker kernel. For example, an application communicating through Entire Net-Work would specify on each command the Broker ID:

```
ETB001::NET
```

This can be abbreviated to the following for the Assembler stubs executing on z/OS (BROKER, CICSETB, COMETB, MPPETB):

```
ETB001
```

where the Entire Net-Work ID of the Broker kernel is 001.

SSL and TLS

Application programs using Secure Sockets Layer (SSL) or Transport Layer Security (TLS) as the transport must specify the SSL settings to the broker stub before any communication with the Broker can take place. There are various methods of setting SSL or TLS transport. See SETSSLPARMS and *Running Broker with SSL or TLS Transport* under z/OS | UNIX | Windows.

Example: An application communicating through SSL or TLS would specify on each command the Broker ID:

```
MYPC:1958:SSL
```

where the host on which the Broker kernel executes is known to SSL or TLS as MYPC and is listening on port 1958.

Transport Examples

- **For programming language C under Windows:**

```
strcpy( pSBuf, "TRUST_STORE=c:\\certs\\CaCert.pem&VERIFY_SERVER=N");
EtbCb.send_length = strlen(pSBuf);
EtbCb.errtext_length = 40;
EtbCb.function = FCT_SETSSLPARMS
rc = broker (etbcb, pSBuf, (char *) 0, pEBuf);
```

- **For programming language Natural under z/OS:**

```
MOVE 'TRUST_STORE=UID/KEYRING' TO #SSL-BUFF
MOVE 80 TO #ETBCB.#SEND-LENGTH MOVE 40 TO #ETBCB.#ERRTEXT-LENGTH
MOVE #FCT-SSLP TO #ETBCB.#FUNCTION
MOVE 'IBMHOST:1958:SSL' TO #ETBCB.#BROKER-ID
...
CALL 'BROKER' #ETBAPI #SSL-BUFF #RECV-BUFF #ERR-TXT
```

See table above for how SSL or TLS is supported depending on broker stub and platform.

For information on Secure Sockets Layer, see *SSL or TLS and Certificates with EntireX*.

Considerations for Writing Applications

- The ACI field `WAIT` allows the application to place the sending or receiving program in a `WAIT` state for a specified time; data or a reply will therefore be received before control is passed to the calling program. When a `WAIT` value is specified for a `SEND` / `RECEIVE` or `RECEIVE_PUBLICATION` function, the calling application waits until the specified time has elapsed or a notification event occurs.

- `WAIT`=`YES` makes additional handling necessary in the Broker stub, whereby `YES` is replaced by the maximum integer value. We recommend you specify a finite value instead of `YES`.

- If frequent outages are expected in the network connections, it is useful to set the transport timeout to *n* seconds. After *n* seconds, the Broker stub terminates the TCP connection, if there is no response from the other side (the Broker kernel). This will help free up the network on the application side. In the case of applications for which the `WAIT` value is specified in the ACI control block (that is, blocking applications), the actual timeout value is the total of the transport timeout plus `WAIT` time.

- TCP/IP only:

    - The Broker ID can contain either an IP address or a hostname. If a hostname is used, it should be a valid entry in the domain name server.

- A LOGOFF call to the Broker kernel will only logically disconnect the application from the Broker kernel. The physical TCP/IP connection is not released until the application terminates.

Restrictions with API Versions 1 and 2

The following maximum message sizes apply to all transport methods:

- ACI version 1: 32167 bytes
- ACI version 2: 31647 bytes

Variable-length Error Text

In previous ACI versions, Broker kernel always returned 40 bytes of error text, space-padded if necessary. For ACI version 9 and above, variable length error text can now be returned if requested. With ACI 9 and above, error text up to the requested length is returned via a new section in the ACI reply. For any previous ACI versions, ETXL is not sent, and the error text is returned by the traditional method.

Note that the error text will continue to be traced in the stub and kernel trace and kernel command log.

See *Broker ACI Fields*.

Programmatically Turning on Command Logging

You can trigger command logging for EntireX components that communicate with Broker by setting the field LOG-COMMAND in the ACI control block.

All functions with LOG-COMMAND programmatically set in the ACI string field will have their commands logged, regardless of any filter settings. Because the LOG-COMMAND option will override any command-log filter settings, remember to reset the LOG-COMMAND field if subsequent requests do not need to be logged.