Writing Applications: Publish and Subscribe

This chapter describes how to implement and program publish-and-subscribe applications - employing durable subscription techniques - with EntireX Broker. Publish-and-subscribe communication is used if data is to be published in order to make it available to one or more subscribers. This communication model is implemented as an independent subsystem in EntireX Broker, that is, it can be activated by setting attributes or left inactive.

For ease of use, we recommend you use the aids and techniques below in the order given.

This chapter covers the following topics:

- Overview of Communication Models
- Basic Concepts of Publish and Subscribe
- API-TYPE and API-VERSION
- LOGON and LOGOFF
- USER-ID and TOKEN
- Control Block Fields and Verbs
- Implementation of Publisher and Subscriber Components
- Blocked and Non-blocked Broker Calls
- Timeout Parameters
- Configuration Prerequisites for Durable Subscriptions
- 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 *Concepts of Persistent Messaging*.

Overview of Communication Models

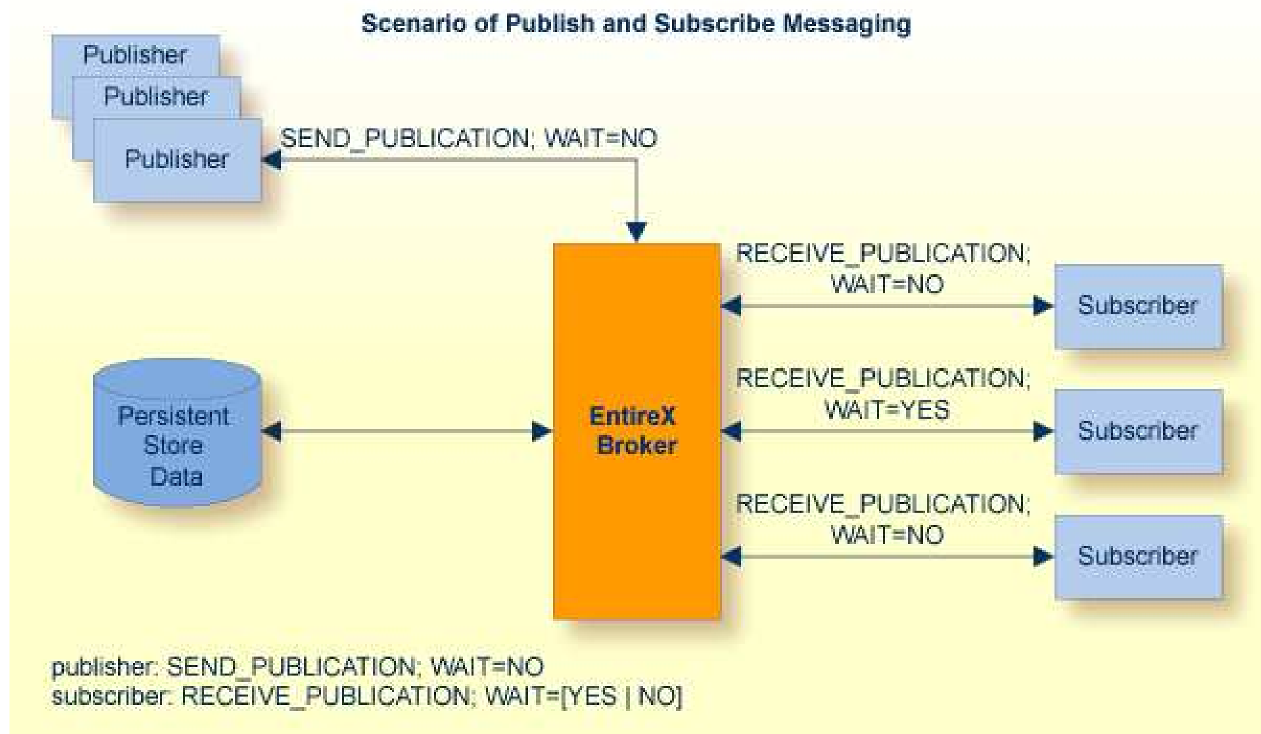
There are two communication models in EntireX Broker: publish and subscribe and client and server.

- **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.

- **Client and Server**
  This communication model is based on a logical connection between exactly two partners: a client and a server. It covers these communication requirements conversationally and non-conversationally, and synchronously and asynchronously. See *Writing Applications: Client and Server*.

Basic Concepts of Publish and Subscribe

- Topic
- Publication
- Subscription
- Publisher
- Subscriber
- Durability of Subscriptions
- Subscription Expiration



Topic

A topic is a logical grouping of publications relating to one subject area, which is defined in the *Broker Attributes*. Topics reflect subject areas, for example current news, stock quotations, weather, online chat, sales systems. Data can be published to a topic only if there are current subscribers to this topic.

Note:
For EntireX Broker, the term "topic" is analogous to the term "service". Topic represents the grouping of related information flows for the publish-and-subscribe communication model, as service does for the client-and-server model.

Publication

A publication is a message or set of messages that are created atomically by one publisher and are available to all current subscribers to the topic. Messages for publication are queued to the topic on a first-in, first-out basis.

Each publication is assigned a unique `PUBLICATION-ID` by EntireX Broker when the publication is created. The `PUBLICATION-ID` is returned to the publisher on the first `SEND_PUBLICATION` command issued when creating a new publication. The `PUBLICATION-ID` is also returned to the subscriber on the first `RECEIVE_PUBLICATION` command used to receive each new publication. Publisher and subscriber must include the `PUBLICATION-ID` for all subsequent commands relating to the same publication.

Subscription

A subscription identifies a user’s intention to receive publications for a specified topic. An active subscription requires the user to have issued a `SUBSCRIBE` command without issuing a subsequent `UNSUBSCRIBE` command. Only publications created after the time of subscription can be read by the subscriber. Conversely, publications created after the `UNSUBSCRIBE` command, or after the subscription has expired, cannot be received by the subscriber, even if the subscription is renewed at a later time. The time period of the subscription determines which publications can be delivered to the subscriber. See *Durability of Subscriptions*.

Publisher

A publisher is a user participating in publish and subscribe that creates publications for one or more topics. It is possible for a publisher to create publications only if there is currently at least one subscription to the topic. This prevents superfluous data from being assigned to the topic.

Subscriber

A subscriber is a user participating in publish and subscribe that can read publications from one or more topics.

Durability of Subscriptions

The behavior of a subscription when the subscriber logs off or broker shuts down is determined by an option specified in the original subscription command.

- **Durable Subscription**

  EntireX Broker enables publish-and-subscribe applications to execute with durable subscriptions by maintaining the user’s subscription status on disk. This ensures that - in the case of a system failure - subscriber information will automatically be recovered, allowing applications to be restarted without any loss of data.

  If `DURABLE` is specified within the `SUBSCRIBE` command, users need only subscribe once to a topic. The subscription is retained after the user issues a `LOGOFF` command or if the subscriber has timed out. Similarly, the subscription remains if the broker is restarted. All publications necessary to satisfy subscription requirements are also retained. If a subscriber no longer wishes to subscribe to a topic, the subscriber must issue an `UNSUBSCRIBE` command; otherwise the subscription remains valid until the subscription expiration time has passed. Durable subscription requires the

administrator to configure the persistent store. See *Concepts of Persistent Messaging*. In addition, the topic must be specified as durable in the *Broker Attributes*.

Durable subscription comprises:

- a list of subscribers and topics to which subscribers have durably subscribed;
- information about the last publication received.

A user has to subscribe only once to a topic. The persistent status remains after the broker is restarted. A subscriber signals its intention to receive publications by issuing a `SUBSCRIBE` command and specifying the topic of interest. If the administrator has specified this topic in the broker's attribute file with a characteristic of `ALLOW-DURABLE`, users will be able to subscribe durably to the topic.

- **Non-durable Subscription**

  Publish-and-subscribe applications can also employ non-durable subscription techniques, if desired. Publications (messages sent from publishers to subscribers) can be either durable or non-durable.

  If durable is not specified in the subscribe command, the subscription is valid only until one of the following events occurs, after which subscription is terminated and publication can no longer be retrieved:

  - the user issues a `LOGOFF` command;
  - Broker is restarted;
  - the subscriber non-activity time value has passed;

    or

  - the subscription expiration time has passed.

The time at which the `SUBSCRIBE` command is issued is significant to the user's subscription. Only publications created after this point in time can be read by the subscriber. Conversely, publications created after either the time at which the `UNSUBSCRIBE` command was issued or the subscription has expired cannot be received by the subscriber, even if the subscription is renewed at a later time. The time period of the subscription determines which publications are delivered.

It is possible for a publisher to create publications only if there is currently at least one subscription to the topic. This feature prevents superfluous data from being assigned to the topic.

Subscription Expiration

A topic is specified in the broker's attribute file with a characteristic of `SUBSCRIPTION-EXPIRATION` time. This is the time period for which the user's subscription remains in effect. After the time period has elapsed, the user's subscription is terminated and the subscription is removed by the broker.

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 broker function is required in order to use the publish-and-subscribe programming model in your application. We recommend that the application issue a LOGOFF function call for the following reasons:

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

Logon 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 with Durable Subscription
- 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 with Durable Subscription

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 API. There are five distinct functions in the Broker ACI which are relevant to publish and subscribe:

- **CONTROL_PUBLICATION**
  The function `CONTROL_PUBLICATION` is used by both the publisher and the subscriber. The publisher uses `CONTROL_PUBLICATION,OPTION=COMMIT` to commit the publication it is creating; the subscriber uses `CONTROL_PUBLICATION,OPTION=COMMIT` to acknowledge the receipt of the publication it is receiving.

- **RECEIVE_PUBLICATION**
  The function `RECEIVE_PUBLICATION` is used by the subscriber to receive all or part of a publication. The field `PUBLICATION-ID` defines the behavior of this function. `RECEIVE_PUBLICATION,PUBLICATION-ID=NEW` signals the subscriber's readiness to obtain the next available new publication, whereas the value `PUBLICATION-ID=`*nnn* specifies that the next message within an existing publication is being requested. After all messages have been received, the publication is acknowledged, using the function `CONTROL_PUBLICATION,OPTION=COMMIT`.

- **SEND_PUBLICATION**
  The function `SEND_PUBLICATION` is used by the publisher to produce a publication. The field `PUBLICATION-ID` defines the behavior of this function. The publisher uses `SEND_PUBLICATION,PUBLICATION-ID=NEW` to create a new publication. The value `PUBLICATION-ID=`*nnn* indicates that a subsequent message within the same publication is being sent, which can be necessary when creating large publications. A publication is completed with the function `SEND_PUBLICATION,OPTION=COMMIT` or with the function call `CONTROL_PUBLICATION`, using the option `COMMIT`.

- **SUBSCRIBE**
  The function `SUBSCRIBE` registers a user with the broker as a subscription for a certain topic. Specifying `SUBSCRIBE,OPTION=DURABLE` determines that the subscription is to be durable. Otherwise the subscription is non-durable.

- **UNSUBSCRIBE**
  The function `UNSUBSCRIBE` covers the opposite functionality: a subscription is cancelled or dissolved.

The participants in publish-and-subscribe are identified by ACI fields `USER-ID` and `TOKEN`.

ACI Syntax

<table>
<tr><th>Function</th><th>Fields in EntireX Broker Control Block</th></tr>
<tr><td>CONTROL_PUBLICATION</td><td>API = 8
, BROKER-ID = BROKER-ID
, USER-ID = user_id
, TOKEN = token
, OPTION = { BACKOUT |
CANCEL |
COMMIT |
LAST |
QUERY |
SETUSTATUS }
[, PUBLICATION-ID = pub_id]
[, USTATUS = user_status]</td></tr>
<tr><td>RECEIVE_PUBLICATION</td><td>API = 8
, BROKER-ID = BROKER-ID
, USER-ID = user_id
, TOKEN = token
, WAIT = NO | YES | wait_value
, PUBLICATION-ID = pub_id | NEW | OLD | ANY
, TOPIC = topic_name</td></tr>
<tr><td>SEND_PUBLICATION</td><td>API = 8
, BROKER-ID = BROKER-ID
, USER-ID = user_id
, TOKEN = token
[, OPTION = COMMIT]
, PUBLICATION-ID = pub_id | NEW
, TOPIC = topic_name
[, USTATUS = user_status]</td></tr>
<tr><td>SUBSCRIBE</td><td>API = 8
, BROKER-ID = BROKER-ID
, USER-ID = user_id
, TOKEN = token
, TOPIC = topic_name
[, OPTION = DURABLE]</td></tr>
<tr><td>UNSUBSCRIBE</td><td>API = 8
, BROKER-ID = BROKER-ID
, USER-ID = user_id
, TOKEN = token
, TOPIC = topic_name</td></tr>
</table>

Key ACI Field Names

The following table lists key ACI field names used to implement applications that use the publish-and-subscribe 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	Description
FUNCTION	Function code for one of the verbs (see table below).
OPTION	Indication of specific broker behavior, depending on the function.
PUBLICATION-ID	Identifier to obtain and specify the publication. Indicates a specific publication. The publication ID value is an internally generated identifier (containing alphanumeric characters) for the publication. We recommend that application programmers make no assumptions about the content, layout or meaning of any part of the PUBLICATION-ID field.
TOPIC	Identifies the name of the publication's topic.
WAIT	Value to specify blocking or non-blocking command.

Key Verbs for FUNCTION Field

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

See *Broker ACI Functions* for all functions.

Verb	Description
CONTROL_PUBLICATION	Publisher uses this to commit and subscriber uses this to acknowledge publications.
RECEIVE_PUBLICATION	Retrieves publication from the broker.
SEND_PUBLICATION	Sends publication to the broker.
SUBSCRIBE	Informs the broker of the existence of a subscriber to a topic.
UNSUBSCRIBE	Informs the broker that the subscriber wishes to unsubscribe.

Implementation of Publisher and Subscriber Components

- Single Message Publication
- Multiple Message Publication

Single Message Publication

This example illustrates a publisher creating single-message publications that are retrieved by one or more subscriber applications. The publisher and subscriber operate asynchronously of each other. There is no reply from the subscriber in this communication model.

This example, which uses durable subscription, shows the typical structure of a subscriber application that has previously subscribed to a topic and is now retrieving the publications issued to that topic. Subscription occurs either during one-time processing provided by the subscriber application, or it is performed explicitly by an administrator. See Broker Command and Information Services.

The subscriber performs `RECEIVE_PUBLICATION` commands in a loop specifying `WAIT=YES`, which makes it possible to process publications as they occur. If none are received during the specified wait period, the server executes another iteration of the loop and repeats the wait until a publication is received.

The `RECEIVE_PUBLICATION` command specifies `PUBLICATION-ID=NEW` to receive all new publications arriving from the publisher. This example assumes single-message publications which do not require acknowledgment of receipt since `AUTO-COMMIT-FOR-SUBSCRIBER=YES` was specified in the topic-specific attributes for the topic NYSE in this case.

```
LOGON USER-ID=SB1,TOKEN=TKSB1
Repeat
    RECEIVE_PUBLICATION,PUBLICATION-ID=NEW,WAIT=YES,TOPIC=NYSE,USER-ID=SB1,TOKEN=TKSB1
        If (Error-Class = 0 and Error-Number = 0)
        /* something received: process request*/

        End-if /* otherwise nothing received */
End-repeat
LOGOFF USER-ID=SB1,TOKEN=TKSB1
```

A publisher issues a `SEND_PUBLICATION` command to send publications containing a single message to a topic. The publisher's `SEND_PUBLICATION` commands are performed with `WAIT=NO`, and `PUBLICATION-ID=NEW` is assigned each time.

```
SEND_PUBLICATION,PUBLICATION-ID=NEW,WAIT=NO,OPTION=COMMIT,TOPIC=NYSE,USER-ID=PB1,TOKEN=TKPB1
```

Multiple Message Publication

This example, which uses durable subscription, shows a publisher creating multiple-message publications that are retrieved by one or more subscriber applications. The publisher and subscriber operate asynchronously of each other; there is no reply from the subscriber in this communication model. In this example, one or more publishers in a stock exchange system send current stock exchange quotations. The subscriber accesses the system at irregular intervals and receives all publications currently available.

This example illustrates the typical structure of a subscriber application that has previously subscribed to a topic and is now retrieving all available publications for a specified topic. Subscription has already occurred either during one-time processing within the subscriber application, or it is performed explicitly by an administrator. See Command-line Utilities.

The subscriber performs RECEIVE_PUBLICATION commands in the outer loop, specifying PUBLICATION-ID=NEW in order to receive the first available publication. The inner loop allows remaining messages within the same publication to be retrieved, after which CONTROL_PUBLICATION acknowledges receipt of the publication. The outer loop is then repeated to obtain the next available publication in conjunction with the inner loop until all available publications are processed.

The RECEIVE_PUBLICATION command specifies PUBLICATION-ID=NEW to receive all new publications. In this case, the subscriber explicitly acknowledges receipt of the publication, using the CONTROL_PUBLICATION function, since it is assumed AUTO-COMMIT-FOR-SUBSCRIBER was not specified in the topic-specific attributes for the topic NYSE in this case.

```
LOGON USER-ID=SB1,TK=TKSB1
While publications available
  RECEIVE_PUBLICATION PUBLICATION-ID=NEW,TOPIC=NYSE,WAIT=YES,USER-ID=SB1,TOKEN=TKSB1
  While data on publication
    RECEIVE_PUBLICATION PUBLICATION-ID=publication-id,TOPIC=NYSE,WAIT=NO,USER-ID=SB1,TOKEN=TKSB1
  End-while
  CONTROL_PUBLICATION OPTION=COMMIT,PUBLICATION-ID=publication-id,TOPIC=NYSE
End-while
LOGOFF USER-ID=SB1,TOKEN=TKSB1
```

A publisher issues a SEND_PUBLICATION command to send a publication containing multiple messages. The publisher’s SEND_PUBLICATION command is performed with WAIT=NO and PUBLICATION-ID=NEW. Remaining messages belonging to this publication are sent to the broker by specifying the generated PUBLICATION-ID within each subsequent SEND_PUBLICATION command. These messages are committed by issuing the CONTROL_PUBLICATION command, which also specifies the generated PUBLICATION-ID.

```
LOGON USER-ID=PB1,TK=TKPB1
SEND_PUBLICATION PUBLICATION-ID=NEW,TOP=NYSE,USER-ID=PB1,TOKEN=TKPB1
While data
  SEND_PUBLICATION PUBLICATION-ID=publication-id,USER-ID=PB1,TOKEN=TKPB1
End-while
CONTROL_PUBLICATION OPTION=COMMIT,PUBLICATION-ID=publication-id,USER-ID=PB1,TOKEN=TKPB1
LOGOFF USER-ID=PB1,TOKEN=TKPB1
```

Blocked and Non-blocked Broker Calls

- Non-blocked Command: WAIT=NO
- Blocked Command: WAIT=YES or WAIT=n

In the publish-and-subscribe communication model, the term "blocked call" refers only to the broker RECEIVE_PUBLICATION command used by subscriber applications. The SEND_PUBLICATION command is always "non-blocking", such that WAIT=NO must be specified. A publisher application sends a publication via EntireX Broker for a specified topic without waiting for any subscribers to receive the publication.

A subscriber application component can use the control block field WAIT in the following ways to determine whether broker will automatically generate a WAIT in order for the command to be either received or satisfied by the partner application:

Non-blocked Command: WAIT=NO

`RECEIVE_PUBLICATION` allows a subscriber application to request a publication for a specified topic. If there are no publications currently available, an ACI response code is returned, indicating that no publications are currently available for the designated topic. See *Error Messages and Codes*. Similarly, a response code also indicates that there are no further messages to be received within the same publication, where `PUBLICATION-ID=`*nnn* has been specified to retrieve continuation segments of the same publication. This technique is used by subscriber applications only.

Example: Subscriber

The subscriber application component requests the next new publication, which is returned if available. If there is no publication available, the subscriber receives a return code immediately, indicating no publications are available at this time. There is no waiting, and the application performs this command periodically under control of the application logic, as shown here:

```
RECEIVE-PUBLICATION,PUBLICATION-ID=NEW,WAIT=NO,TOPIC=NYSE,USER-ID=SB1,TOKEN=TKSB1
... application code to process publication ....
```

Blocked Command: WAIT=YES or WAIT=n

Allows a subscriber application to solicit a publication to be returned for the specified topic. The calling application is automatically placed in a `WAIT` state until there is a publication available for the specified topic. If no publication is available during the specified waiting time, an ACI response code is returned to the application, indicating that no publications are currently available for the designated topic. See *Error Messages and Codes*. Similarly, a response code also indicates that there are no further messages to be received within the same publication if `PUBLICATION-ID=`*nnn* has been specified in order to retrieve continuation segments of the same publication. This technique is used by subscriber applications only.

Example: Subscriber

The subscriber application component requests the next new publication, which is returned if available. If there is no publication available, the subscriber enters a `WAIT` state for the specified (or default) time period, during which it is eligible to receive any new publications that arrive in this time. At the end of the specified (or default) time period, the subscriber receives a return code if no publications were available. The following example shows this process being repeated indefinitely within a loop:

```
Repeat
    RECEIVE-PUBLICATION,PUBLICATION-ID=NEW,WAIT=YES,TOPIC=NYSE,USER-ID=SB1,TOKEN=TKSB1
    ... application code to process publication ....
End-repeat
```

Timeout Parameters

- Timeout Behavior
- Types of Non-activity Time
- Recommendations

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 `PUBLISHER-NONACT` and `SUBSCRIBER-NONACT` attributes are non-activity timeout parameters which can be specified independently of each other to control the timeout behavior of publisher and subscriber application components. If an application component issues no commands to the broker for the specified time period, the broker logs the user off automatically, cleaning up related in-memory resources. See *LOGON and LOGOFF*. If the subscriber did not issue a durable `ALLOW-DURABLE` command, the user's subscription will also be removed after this time.
- The `SUBSCRIPTION-EXPIRATION` attribute determines the lifetime of a user's durable subscription. Durable subscriptions are retained by the broker until either the subscriber issues an `UNSUBSCRIBE` command or the subscription lifetime has expired.
- The `PUBLICATION-LIFETIME` attribute determines how long publications are retained by the broker until they are either received by all subscribers or the publication lifetime has expired.
- The `WAIT` field in the ACI control block is significant only to the subscriber application component. The program is placed into a `WAIT` state for a specified time when issuing the `RECEIVE_PUBLICATION` command, allowing 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 making a blocked command. A non-blocked command is executed if `WAIT=NO` is specified. See *Blocked and Non-blocked Broker Calls*. The `SEND_PUBLICATION` command is always issued with `WAIT=NO`.

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`
- `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 publish-and-subscribe applications:

- If the subscriber issues blocked `RECEIVE_PUBLICATION` commands, make the `WAIT` time adjustable. The `WAIT` value can be read as a startup parameter from the user-written INI or CFG file, or any other parameter data set or set of environment variables, depending on the platform.

- When using non-durable subscriptions, ensure the specified `SUBSCRIBER-NONACT` time is not exceeded by the subscriber between issuing commands to the broker; otherwise the user will be automatically logged off, and the user's subscription will be removed during a period of inactivity. See *Durability of Subscriptions*

  Note:
  When blocking `RECEIVE_PUBLICATION` commands, the `SUBSCRIBER-NONACT` value is overridden by the `WAIT` time (if this is greater).

- If there are no available messages for the duration of a blocked `RECEIVE_PUBLICATION` command, response code 00740074 is returned to the subscriber. The subscriber can reissue the `RECEIVE_PUBLICATION` command repeatedly until the next publication becomes available. See also *Blocked Command: `WAIT=YES` or `WAIT=n`*.

- If there are no available messages when issuing a non-blocked `RECEIVE_PUBLICATION`, the command returns response code 00030488. See also *Non-blocked Command: `WAIT=NO`*.

- If there are no further messages available when issuing a `RECEIVE_PUBLICATION`, the command returns response code 00740480.

- Ensure that the `PUBLISHER-NONACT` time is not exceeded by the publisher between issues of `SEND_PUBLICATION` commands; otherwise the user will be automatically logged off, and any unfinished or uncommitted publications will be lost.

Configuration Prerequisites for Durable Subscriptions

A subscription can be durable or non-durable. See *Durability of Subscriptions*. Durable subscriptions require additional configuration steps. Since subscriber information for durable subscriptions must also be present after a broker is restarted, a persistent store is required (`PSTORE`). See also *Concepts of Persistent Messaging*. This allows Adabas (all platforms), file system (UNIX and Windows) and DIV (z/OS) to be utilized for storing both publication information and, optionally, subscription information.

If you use the persistent store for subscriber information under Adabas, see *Configuring and Operating the Adabas Persistent Store* under z/OS | UNIX | Windows | z/VSE. If you are using persistent store type DIV or the local file system, no additional PSTORE configuration is required. See also *Broker-specific Attributes* under *Broker Attributes* for other related parameters.

```
NUM-TOPIC-TOTAL          = 4
NUM-SUBSCRIBER-TOTAL     = 8
SUBSCRIBER-STORE         =PSTORE
```

Note:
The topic attribute definitions must specify `ALLOW DURABLE=YES`. Otherwise durable subscription requests are rejected.

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.

<table>
<tr><th rowspan="2">Operating System</th><th rowspan="2">Environment</th><th rowspan="2">Module</th><th colspan="4">Transport to Broker</th></tr>
<tr><th>TCP</th><th>SSL</th><th>NET (1)</th><th>HTTP(S) (6)</th></tr>
<tr><td rowspan="7">z/OS (2)</td><td>Batch, TSO, IMS (BMP)</td><td>BROKER</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>Com-plete</td><td>COMETB</td><td>x</td><td>(3)</td><td>x</td><td></td></tr>
<tr><td>CICS</td><td>CICSETB</td><td>x</td><td>(3)</td><td>x</td><td></td></tr>
<tr><td>IMS (MPP)</td><td>MPPETB</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>IDMS/DC (4)</td><td>IDMSETB</td><td>x</td><td>(3)</td><td></td><td></td></tr>
<tr><td>Natural</td><td>NATETB23</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>UNIX System Services</td><td>Java ACI</td><td>x</td><td>x</td><td></td><td>x</td></tr>
<tr><td colspan="2" rowspan="2">UNIX</td><td>broker.so</td><td>x</td><td>x</td><td></td><td></td></tr>
<tr><td>Java ACI</td><td>x</td><td>x</td><td></td><td>x</td></tr>
<tr><td colspan="2" rowspan="2">Windows</td><td>broker.dll (5)</td><td>x</td><td>x</td><td></td><td></td></tr>
<tr><td>Java ACI</td><td>x</td><td>x</td><td></td><td>x</td></tr>
<tr><td>BS2000/OSD</td><td>Batch, Dialog (formerly TIAM)</td><td>BROKER</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td rowspan="2">z/VSE</td><td>Batch</td><td>BKIMB</td><td>x</td><td></td><td>x</td><td></td></tr>
<tr><td>CICS</td><td>BKIMC</td><td>x</td><td></td><td>x</td><td></td></tr>
<tr><td colspan="2">z/VM</td><td>BKIMBCMS</td><td>x</td><td></td><td>x</td><td></td></tr>
<tr><td colspan="2">IBM i</td><td>EXA</td><td>x</td><td></td><td></td><td></td></tr>
<tr><td colspan="2">OpenVMS</td><td>BROKER</td><td>x</td><td>x</td><td></td><td></td></tr>
</table>

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.