BEA MessageQ

Programming Guide

BEA MessageQ for OpenVMS Version 5.0
Document Edition 5.0
March 2000

BEA MessageQ Programmer's Guide

Document Edition	Date	Software Version
5.0	March 2000	BEA MessageQ, Version 5.0

Contents

Preface

Purpose of This Document

This document provides a detailed description about using the BEA MessageQ application programming interface (API) to build and integrate distributed applications. This document contains both tutorial and reference information.

Who Should Read This Document

This document is intended for applications designers and developers who are interested in designing, developing, building, and running BEA MessageQ applications.

How This Document Is Organized

The BEA MessageQ Programmer’s Guide is organized as follows:

- Chapter 1, “Sending and Receiving BEA MessageQ Messages” describes the basic process of sending and receiving messages. This chapter makes a distinction between sending messages as predefined message buffers and as self-describing FML buffers.
- Chapter 2, “Using Recoverable Messaging” describes how to guarantee message delivery using recoverable messages written to nonvolatile storage.
- Chapter 3, “Broadcasting Messages” describes how to send messages to multiple queues with a single program call using Selective Broadcast Services.

- Chapter 4, “Using Naming” describes how to enable BEA MessageQ applications to identify message queues by name.
- Chapter 5, “Using Message-Based Services” describes how to use message-based services to obtain the status of one or more queues, monitor and control link status, and broadcast messages.
- Chapter 6, “Building and Testing Applications”describes how to format and convert message data, write portable BEA MessageQ applications, test and debug applications, and control message flow.
- Chapter 7, “Using the Script Facility” describes how to simulate message exchange between programs using script files instead of test programs.
- Chapter 8, “PAMS Application Programming Interface” describes the BEA MessageQ applications programming interface. All API functions are listed in alphabetical order.
- Chapter 9, “Message Reference” describes all BEA MessageQ message-based services.
- Appendix A, “Supported Delivery Modes and Undeliverable Message Actions,” lists the supported combinations of delivery mode (such as wait for completion or no notification) and undeliverable message action (such as discard, return to sender, or store and forward).
- Appendix B, “Obsolete Functions and Services,” lists obsolete BEA MessageQ API functions and message-based services. These items are listed for use with applications built on older versions of BEA MessageQ and should not be used for new development.

How to Use This Document

This document, BEA MessageQ Programmer’s Guide, is designed primarily as an online, hypertext document. If you are reading this as a paper publication, note that to get full use from this document you should access it as an online document via the BEA MessageQ Online Documentation CD.

The following sections explain how to view this document online, and how to print a copy of this document.

Opening the Document in a Web Browser

To access the online version of this document, open the `index.htm` file in the top-level directory of the BEA MessageQ Online Documentation CD. On the main menu, click the BEA MessageQ Programmer's Guide button.

Note: The online documentation requires a Web browser that supports HTML Version 3.0. Netscape Navigator version 3.0 or Microsoft Internet Explorer version 3.0 or later are recommended.

Printing from a Web Browser

You can print a copy of this document, one file at a time, from the Web browser. Before you print, make sure that the chapter or appendix you want is displayed and *selected* in your browser.

To select a chapter or appendix, click anywhere inside the chapter or appendix you want to print. If your browser offers a Print Preview feature, you can use the feature to verify which chapter or appendix you are about to print. If your browser offers a Print Frames feature, you can use the feature to select the frame containing the chapter or appendix you want to print. The BEA MessageQ Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document. On the CD's main menu, click the Bookshelf button. On the Bookshelf, scroll to the entry for the BEA MessageQ document you want to print and click the PDF option.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.

Convention	Item
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <stdio.h>`
	`void main ( )`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: **`dmqshutdown`** `-b` *`bus_id`* `-g` *`group_id [-f]`*
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `dmqshutdown -b` *`bus_id`* `-g` *`group_id [-f]`*

Convention	Item
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `dmqshutdown -b bus_id -g group_id ...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Related Documentation

The following sections list the documentation provided with the BEA MessageQ software, related BEA publications, and other publications related to the technology.

BEA MessageQ Documentation

The BEA MessageQ information set consists of the following documents:

BEA MessageQ Installation and Configuration Guide for Windows NT

BEA MessageQ Installation and Configuration Guide for UNIX

BEA MessageQ Installation Guide for OpenVMS

BEA MessageQ Configuration Guide for OpenVMS

BEA MessageQ Programmer's Guide

BEA MessageQ FML Programmer's Guide

BEA MessageQ Reference Manual

BEA MessageQ System Messages

BEA MessageQ Client for Windows User's Guide

BEA MessageQ Client for UNIX User's Guide

BEA MessageQ Client for OpenVMS Guide

Note: The BEA MessageQ Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document.

Contact Information

The following sections provide information about how to obtain support for the documentation and software.

Documentation Support

If you have questions or comments on the documentation, you can contact the BEA Information Engineering Group by e-mail at **docsupport@beasys.com**. (For information about how to contact Customer Support, refer to the following section.)

Customer Support

If you have any questions about this version of WebLogic Enterprise, or if you have problems installing and running WebLogic Enterprise, contact BEA Customer Support through BEA WebSupport at `www.beasys.com`. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 Sending and Receiving BEA MessageQ Messages

This chapter covers the following topics:

- The Basics of Sending and Receiving Messages
- Sending and Receiving Message Buffers
- Receiving Messages Using Message Pointers
- Self-Describing Messaging with FML
- Exchanging Messages Between BEA MessageQ and BEA TUXEDO or BEA M3

Overview

BEA MessageQ enables applications to exchange information in the form of messages using the following PAMS API functions:

- The `pams_attach_q` function—attaches the application to the message queuing bus and defines a queue for the application to receive messages
- The `pams_put_msg` function—sends a message to a target queue

- The `pams_get_msg` function—retrieves a message from a queue
- The `pams_detach_q` function—detaches the application from the message queuing bus

BEA MessageQ provides applications with three distinct ways to send and receive messages using:

- Static message buffers
- Pointers to message buffers that can be dynamically reallocated as required
- Self-describing messaging using Field Manipulation Language (FML)

This variety of methods for sending and receiving messages enables application developers to choose the type of messaging that best suits the application's present and future needs.

The Basics of Sending and Receiving Messages

To send or receive messages, an application must be attached to at least one message queue on the message queuing bus. This queue serves as the application's primary queue—the main mailbox in which it receives information. To attach to a queue, the application must successfully execute the `pams_attach_q` function. Once attached, the application can send a message to a known target queue address using the `pams_put_msg` function.

BEA MessageQ offers the following functions for receiving messages:

- The `pams_get_msg` function—retrieves a single message from a queue
- The `pams_get_msgw` function—retrieves a single message from a queue but, if the queue is empty, this function waits for a message to arrive in the queue
- The `pams_get_msga` function—asynchronously retrieve messages from a queue. This function is available only on OpenVMS systems.

When the application is finished sending or receiving messages, it detaches from the message queuing bus using the `pams_detach_q` function.

Refer to the programming examples distributed as part of the BEA MessageQ kit to view sample programs for each of these PAMS API functions.

Note: If you are new to using BEA MessageQ, you should begin by reading the *Introduction to Message Queuing*. This introduction explains the BEA MessageQ concepts that you need to understand before you can begin successfully developing applications.

Sending and Receiving Message Buffers

Sending and receiving information as static message buffers is the easiest way to exchange information using BEA MessageQ. A static message buffer is a predefined, static data structure. Often, an application uses a version number to identify the structure layout. So, for example, when a payroll system sends employee payroll information using version 1 of its payroll data structure, the receiving application can interpret each field of data in the buffer because it knows the definition of the version 1 payroll data structure.

Passing information using a static data structure in the form of a message buffer is the fastest way to exchange information between BEA MessageQ applications. Because the data structure definition is known to both the sending and receiving applications, no interpretation is required. Therefore, processing of information between both sender and receiver programs is faster.

See the following topics for information on how to:

- Send a message buffer up to 32K. See the How to Send BEA MessageQ Messages topic for more information.
- Send a message buffer up to 4MB. See the How to Send Large Messages topic for more information.

How to Send BEA MessageQ Messages

When programming BEA MessageQ applications, there are four basic functions that are used in the sending messages. The first function called is `pams_attach_q`. This function is used to connect your BEA MessageQ applications to the BEA MessageQ message queuing bus. Attaching to the message queuing bus provides the application with a default queue address for receiving the reply message and a means to share information with all other BEA MessageQ applications.

The example in Listing 1-1 illustrates how to attach to a queue by name. The queue name must be defined appropriately in your group initialization file.

Listing 1-1 Example of Attaching to a Queue by Name

```
#include <stdio.h>
#include <string.h>
#include "p_entry.h"
#include "p_return.h"
#include "p_symbol.h"
   .
   .
   .
   int32       attach_mode;
   int32       dmq_status;
   int32       q_name_len;
   int32       q_type;
   char        q_name[12];
   q_address   my_primary_queue;
   strcpy(q_name,"example_q_1");
   attach_mode = PSYM_ATTACH_BY_NAME;
   q_type      = PSYM_ATTACH_PQ;
   q_name_len  = (int32)sizeof( q_name );
   dmq_status  = pams_attach_q(
                     &attach_mode,
                     &my_primary_queue,
                     &q_type,
                     q_name,
                     &q_name_len,
                     (int32 *) 0,    /*  Use default name space */
                     (int32 *) 0,    /*  No name space list len */
                     (int32 *) 0,    /*  Timeout                */
                     (char *) 0,     /*  Reserved by MessageQ   */
                   (char *) 0 );    /*  Reserved by MessageQ     */
```

```
if ( dmq_status == PAMS__SUCCESS )
  printf( "Attached successfully to queue: \"%s\".\n", q_name );
else
  printf( "Error attaching to queue: \"%s\"; status returned
             is: %ld\n", q_name, dmq_status );
 .
 .
 .
```

After attaching to a queue, the application uses the `pams_put_msg` function to send a message to the queue address of the receiver program. Before the message can be sent, the application needs to provide application data in a message buffer. The data structure of the message buffer is predefined so that both the sending and the receiving application can interpret the message contents.

The example in Listing 1-2 illustrates how to send a number of messages to a queue.

Listing 1-2 Example of Sending Messages to a Queue

```
    int32         attach_mode;
    int32         dmq_status;
    int32         q_name_len;
    int32         q_type;
    int32         timeout;
    short         class;
    short         type;
    short         msg_size;
    char          delivery;
    char          priority;
    char          uma;
    static char   msg_area[18];
    static char   q_name[12];
    q_address     my_queue;
    struct PSB    put_psb;
    .
    .
    .
    /*
    **  Put a message into my own queue
    */
    priority   = 0;
    class      = 0;
    type       = 0;
    delivery   = PDEL_MODE_NN_MEM;
    msg_size   = (short) strlen( msg_area );
    timeout    = 50;  /* 5 seconds  */
    uma        = PDEL_UMA_DISCL;

    dmq_status = pams_put_msg(
                      msg_area,
                      &priority,
                      &my_queue,
                      &class,
                      &type,
                      &delivery,
                      &msg_size,
                      &timeout,
                      &put_psb,
                      &uma,
                      (q_address *) 0,
                      (char *) 0,
                      (char *) 0,
                      (char *) 0 );

   if ( dmq_status == PAMS__SUCCESS )
      printf( "\n\tPut successfully to queue: \"%s\".\n", q_name );
   else
      printf( "\nError sending to queue: \"%s\"; status returned
                 is: %ld\n", q_name, dmq_status );
    .
    .
    .
```

BEA MessageQ applications use the `pams_get_msg` function to read messages from a queue. Because both sending and receiving programs use the predefined buffer structure, the receiving application can interpret the message.

When a BEA MessageQ application is finished, the `pams_detach_q` is called to disconnect the program from the message queuing bus.

Static data structures limit the flexibility of applications to adapt to changing business conditions. To change the data structure, both the sender and receiver programs must be recoded to send and interpret the new message correctly. In addition, all production applications must be shut down and the newer versions started up for the change to take affect. Such large changes to an integrated application environment often result in synchronization problems where some applications have not yet been restarted using the new message format. This leads to processing errors until all applications are using the same version of the message data structure.

Another limitation in using static message buffers is that data is passed “as is” from one system to another in the network. So, if a message must be delivered between two computers that use different byte orders, the application must perform the byte order translation to ensure that the data is interpreted properly by the target application. BEA MessageQ does not perform data marshalling between systems with unlike hardware data formats when messages are sent using the static message buffer approach.

Prior to BEA MessageQ Version 4.0, the only way to send a message was to use the predefined message data structure which allowed messages to be as large as 32 kilobytes. If either the sending or receiving data structure needed to change the message structure, both sending and receiving applications were programmed to use the new message structure. For this change to take effect, both sending and receiving programs needed to be reloaded.

How to Send Large Messages

BEA MessageQ enables applications to send buffer-style and FML-style messages up to 4MB in size. For FML-style messaging no differences in approach are required to send small or large messages. However, use the following procedure when sending and receiving buffer-style messages larger than 32K.

To send a large buffer-style message, applications still use the `pams_put_msg` function. Most arguments to this call are specified in the same way for large and small messages. However, the following list describes the arguments that are specified differently for large messages:

- the `msq_size` argument must contain the symbol `PSYM_MSG_LARGE` indicating that this is a large message
- the `large_size` argument supplies the size of the large message buffer

To retrieve a large buffer-style message from a queue, you still use the `pams_get_msg`, `pams_get_msgw`, or the `pams_get_msga` functions. To retrieve a large message from an auxiliary journal, use the `pams_read_jrn` function. The following arguments are supplied to these functions to read large messages:

- the `msq_area_len` argument must contain the symbol `PSYM_MSG_LARGE` indicating the operation will return a large message
- the `large_area_len` argument supplies the size of the message buffer to receive the large message

These functions return the actual size of the message written to the message buffer in the `large_size` argument.

A sample program illustrating how to send a large message called `x_putbig.c` is contained in the programming examples directory of your media kit.

Receiving Messages Using Message Pointers

Receiving applications can use message pointers to allow for automatic buffer reallocation when the buffer received is larger than the buffer allocated. (Message pointers are also required for processing self-describing messages based on FML buffers. See Self-Describing Messaging with FML for more information.)

To retrieve a buffer-style message from a queue using `pams_get_msg` and pointers:

- the `msg_area_len` argument must contain the symbol `PSYM_MSG_BUFFER_PTR`
- the `msg_area` argument must point to a pointer to dynamically allocated space or be set to point to a NULL pointer
- the `large_area_len` argument must contain the size of the space allocated for the message or be set to 0 if it is NULL.

If the message received will not fit in the allocated space or if the pointer is NULL, the buffer is reallocated, the pointer to the new buffer is returned in the `msg_area`, and its length is returned in the `large_area_len` arguments.

When the message is retrieved from the queue:

- the message is placed in the buffer referenced by the pointer contained in `msg_area`
- the actual length of the buffer is returned in the `large_size` argument
- the `len_data` argument is set to `PSYM_MSG_BUFFER_PTR`
- the `endian` field in the `show_buffer` structure is set to the appropriate byte ordering scheme for the type of data
- the `large_area_len` argument is updated with the new buffer size if the buffer was reallocated

Self-Describing Messaging with FML

Self-describing messaging using Field Manipulation Language (FML) is new in BEA MessageQ Version 5.0. FML-based messaging replaces the SDM capabilities provided in BEA MessageQ V4.0. While basic information on FML is included in this document, see the *BEA MessageQ FML Programmer's Guide* and the *BEA MessageQ Reference Manual* for more information on FML.

FML is a set of C language functions for defining and manipulating storage structures called fielded buffers, that contain attribute-value pairs in fields. The attribute is the field's identifier, and the associated value represents the field's data content.

Using FML, applications construct messages containing both the message content and the information needed by the receiver program to understand what is in the message. The receiver program dynamically interprets the contents of the message by "decoding" some or all of the data contained in it. Message pointers are used when a receiving application retrieves an FML-style message from a message queue.

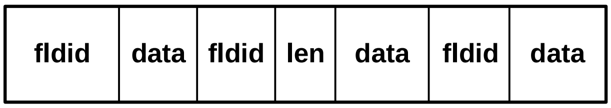
Using FML buffers, applications do not interact with a message structure. Instead, sender programs encode the contents of the message using the appropriate FML function. Each field in the message has a value (the content) and a tag (identifier). When an application retrieves an FML message, the content is not directly visible. The receiver program must use FML functions to interpret the contents of the message that are appropriate to its operation.

Because FML messages contain information about how to interpret the message contents, self-describing messaging provides applications with more flexibility in adding fields to a message or changing the message contents without necessarily needing to recode all of the receiving applications. In addition, FML performs data marshaling of data formats between computer systems with unlike hardware data formats.

How Self-Describing Messaging Works

FML messages, which are accessed by a pointer, contain tagged values that are manipulated by specific FML functions. When you code, you build the message buffer using assignments inside the message data structure which you have defined. FML uses the following fielded buffer structure:

Figure 1-1 Fielded Buffer Structure



In the above figure, the message structure contains pairs of attributes and values. Each field is labeled with an integer that combines information about the data type of the accompanying field with a unique identifying number. The label is called a field identifier or `fldid`. For variable-length items, `fldid` is followed by a length indicator. The buffer can be represented as a sequence of `fldid`/data pairs or `fldid`/length/data triples for variable-length items.

Benefits of Using FML

There are several advantages to using FML. These advantages are as follows:

- Scalability—FML messages can evolve as your business grows. For example, you can add fields to your message in a completely backwards compatible manner. You only need to modify those applications which need the new information. You do not have to change application code that does not need the new information.
- Flexibility—you can change the size of a field at any time without changing an FML application because this type of information is encoded into the message.
- Portability of messages—you do not have to write data transformation routines to handle differences between data types and platforms. FML automatically performs the data transformation for you. The transformations included in FML are network byte order, C data types, word sizes, word alignment, and IEEE floating point.
- Reusability of messages—a single message can be interpreted by several applications that need different parts of the message. For example, suppose a user application needs a person's address and another user application needs the person's hourly wage. Instead of the server application constructing a unique message for each application, it can construct a single message which contains both the person's address and hourly wage. When one of the user applications interprets the message, only the information that is needed by that application is decoded. The other user application can reuse the same message to get only the information that it needs.

FML manages data transformation so that an FML message can be interpreted properly on any platform. Figure 1-1 illustrates how using fielded buffers creates a formatted message that replaces all platform-dependent compiler assignments through an API, which has decoupled and hidden all the machine, operating system, and platform dependencies. It has also properly encoded the message so that it can be safely transported from platform to platform in a heterogeneous environment. Furthermore, it protects applications from message structure changes.

For example, suppose you have an application running on a Hewlett-Packard machine and a Compaq machine and the message data has a little endian data format. When messages are sent to the Compaq machine from the Hewlett-Packard machine, a conversion from little endian to big endian data format must take place. This is handled by encoding the little endian format and converting it to a platform independent format. Then, the platform independent format is decoded into the big endian format for the Compaq machine.

Performance Considerations When Using FML

One performance consideration in using FML is that it uses a larger message size to deliver the same amount of user data and can take longer to pass back and forth between machines. The message size is larger because the message contains both the information and a description of the information, encoded in a platform-independent manner.

For example, consider a message that is 100 characters. With a defined message buffer, the message is only 101 bytes using a C message structure. In a worst case scenario, the FML message size could be 800 bytes. Each of the original 100 bytes requires 1 byte of data and 4 bytes of identifier. Because each byte of data must be aligned on word boundaries for platform independence, each byte requires three additional padding bytes.

A more efficient way to encode character data is to use an array. You can encode the 100 bytes as an array of 100 bytes. With an array, the padding necessary to accomplish word alignment is not needed and the tag is present only once. Using this approach, the actual size needed is 108 bytes (including the tag and length).

You may be able to structure the application to use the larger FML message only when needed and a message buffer at other times. For more information on this technique, see the Designing Applications to Use a Mixed Messaging Environment topic.

An additional performance consideration is the time required to encode and decode information when exchanging messages between platforms having different data formats.

Designing Applications to Use a Mixed Messaging Environment

A mixed messaging environment is an environment where you want to exchange static buffer messages and FML messages in the same application. If you are programming an application to use both kinds of messages, consider having your application use two queues—one queue for buffer-style messages and another queue for FML messages. By designing your application this way, you guarantee that your application does not dequeue an FML message by mistake.

Note that for performance reasons, it might be better to modify the buffer structure and redistribute all software than to use a mixed messaging environment. This may be the recommended approach when your applications are close geographically and there is a convenient time to update software.

How to Send an FML Message

When sending FML messages, you code in a similar manner as with a message buffer. However, the main difference is that messages are manipulated using message pointers rather than using the actual message buffer. The message pointer is provided to `pams_put_msg` as the first argument (`msg_area`). To code an FML message, you must add the following steps to your program logic after attaching to a queue:

1. Define field identifiers and map them to field names.
2. Build messages using the appropriate FML functions.
3. Send the message. To use an FML message pointer when sending a message, the sender program specifies the symbol `PSYM_MSG_FML` as the `msg_size` argument in the `pams_put_msg` function.
4. Once your application is done using the FML message, delete the FML message using `Ffree32()` to prevent memory leaks.

A sample program called `x_fml.c` which illustrates how to send and receive FML messages is distributed as part of your media kit.

Defining Field Identifiers

FML message fields are tagged with field identifiers. Each tag implicitly defines the data type of the information it is associated with. This guarantees that the sender and the receiver of an FML message have an explicit agreement about the kind of information they exchange. The collection of tags builds a kind of message dictionary.

The following table describes the tag data type symbols as defined in `fml32.h`:

Data Type	Symbol
short int	`FML_SHORT`
long int	`FML_LONG`
character	`FML_CHAR`
single-precision float	`FML_FLOAT`
double-precison float	`FML_DOUBLE`
string, null terminated	`FML_STRING`
character array	`FML_ARRAY`

Fields are usually referred to by their field identifier (`fldid`), an integer. This allows you to reference fields in a program without using the field name.

Identifiers are assigned (mapped) to field names in the following ways:

- dynamically at run time using field table files
- statically at compile time using C language header (`#include`) files

A typical application may use one or both of these methods.

Building the FML Message

The FML API provides functions to place tagged values in a fielded buffer accessed with its pointer. A variety of functions are provided to support a large number of buffer operations.

Any field in a fielded buffer can occur more than once. Many FML functions take an argument that specifies which occurrence of a field is to be retrieved or modified. If a field occurs more than once, the first occurrence is numbered 0, and additional occurrences are numbered sequentially.

The example in Listing 1-3 shows a program which builds a message with the queue id and time stamp. The message is then put into a message queue.

Listing 1-3 Example of Building a Fielded Buffer

```
/* applications fields */
#include myFields.h”
FBFR32 *fbfr;
fbfr = Falloc32(10,100);
Fadd32(fbfr, QID, 0, &qid, 0);
Fadd32(fbfr, TSTAMP, 0, &timestamp, 0);
```

Note that FML provides data transparency. That is to say that your application does not know nor need to know how any data values are stored in the message. The FML and PAMS API functions handle this for your application.

Sending the FML Message

After creating a pointer and building the message, you can send the message to the target queue. To send an FML buffer, the sender program specifies the symbol `PSYM_MSG_FML` as the `msg_size` argument to the `pams_put_msg` function. The system verifies that the buffer is an FML32 buffer. If the buffer is not an FML32 buffer, the `pams_put_msg` call will fail and return `PAMS__NOTFLD`.

The code fragment example in Listing 1-4 sends the FML message. The previously encoded message is contained in the `msg_area` argument.

Listing 1-4 Example of Sending an FML message

```
/*  Sends the message identified by the pointer. The symbol   */
/*  PSYM_MSG_FML_ in the msg_size argument indicates that   */
/*  the message is a pointer to an FML buffer.     */

/*  Define any variables needed to the put function here. */
```

```
    msg_size = PSYM_MSG_FML;
.
.
.
    dmq_status = pams_put_msg(
                 (char *) fbfr,
                 &priority,
                 &my_queue,
                 &class,
                 &type,
                 &delivery,
                 &msg_size,
                 &timeout,
                 &put_psb,
                 &uma,
                 (q_address *) 0,
                 (char *) 0,
                 (char *) 0,
                 (char *) 0, );

    If ( dmq_status == PAMS__SUCCESS )
        printf ( “Message pointer successfully put to the queue”);
    else
        printf ( “Error putting message to queue”);
.
.
.
```

How to Receive an FML Message

When receiving FML messages, you code in a similar manner as with a buffer-style message. However, you must add the following steps to your program logic after attaching to a queue:

1. Include the predefined field identifier definitions to your code to guarantee that both sending and receiving applications are using the same definitions.

2. Create a pointer to a pointer to dynamically allocated space using `Falloc` or `malloc` and `Finit`.

3. Set `large_area_len` to the length of the allocated space or to 0 if it is NULL.

4. Read the message from the queue. The receiver program determines whether the message is a pointer to an FML buffer pointer by reading the endian field in the `show_buffer` argument of the `pams_get_msg` or `pams_get_msgw` function. If this field contains the symbol `PSYM_FML`, the message is an FML buffer.
5. Access the message fields using the appropriate FML API functions.
6. Delete or reuse the message pointer to prevent memory leaks.

Note: When an FML message is received, the endian field of the `show_buffer` argument returned by the `pams_get_msg` or `pams_get_msgw` functions is set to `PSYM_FML`.

Reading the Message from the Queue

To read a message from a queue, use the `pams_get_msg` function after you have included the tag definitions and created a message pointer. The code fragment example in Listing 1-5 creates a message handle and gets the message:

Listing 1-5 Example of Reading an FML Message

```
/*  Include the predefined field identifier definition        */

    #include “myfields.h”;
    FBFR32 *fbfr;
    FBFR32 **pfbfr;

/*  Read the message identified by the pointer. The symbol    */
/*  PSYM_MSG_BUFFER_PTR in the len_data argument indicates that   */
/*  the message is a pointer and not a message buffer.     */


/*  Define any variables needed for the get function here. */

    len_data = PSYM_MSG_BUFFER_PTR;
    pfbfr = &fbfr;
.
.
.
    dmq_status = pams_get_msg(
                 (char *) pfbfr,
                 &priority,
                 &msg_source,
                 &class,
                 &type,
                 &msg_area_len,
                 &msg_len,
                 (int32 *)&sel_filter,
                 (struct PSB *) 0,
                 (struct show_buffer *) 0,
                 &show_buffer_len
                 &large_area_len,
                 &large_size,
                 (char *) 0, );

    If ( dmq_status == PAMS__SUCCESS )
        printf ( “Message pointer successfully read”);
printf ( “Error reading message”);
.
.
.
```

Interpreting the Message

After your application creates a message pointer and gets the message, it can interpret the message. Your application can use FML API functions to manipulate the fielded buffer.

Exchanging Messages Between BEA MessageQ and BEA TUXEDO or BEA M3

BEA MessageQ V5.0 include a messaging bridge that allows the exchange of messages between BEA MessageQ V5.0 and BEA TUXEDO V6.4 or BEA M3 2.1. BEA MessageQ applications can send a message using `pams_put_msg` that a TUXEDO application can retrieve through a call to `tpdequeue`. TUXEDO applications can send a message using `tpenqueue` that a BEA MessageQ application can retrieve through a call to `pams_get_msg(w)`. In addition, a BEA MessageQ application can invoke a TUXEDO service using `pams_put_msg`. It is also possible for a TUXEDO application to use `tpenqueue` to put a message on a queue and to use `tpdequeue` to retrieve a message from a queue.

This exchange of messages is made possible by two TUXEDO servers that are included in the BEA MessageQ kit and that run on the same machine as BEA MessageQ: TMQUEUE_BMQ and TMQFORWARD_BMQ.

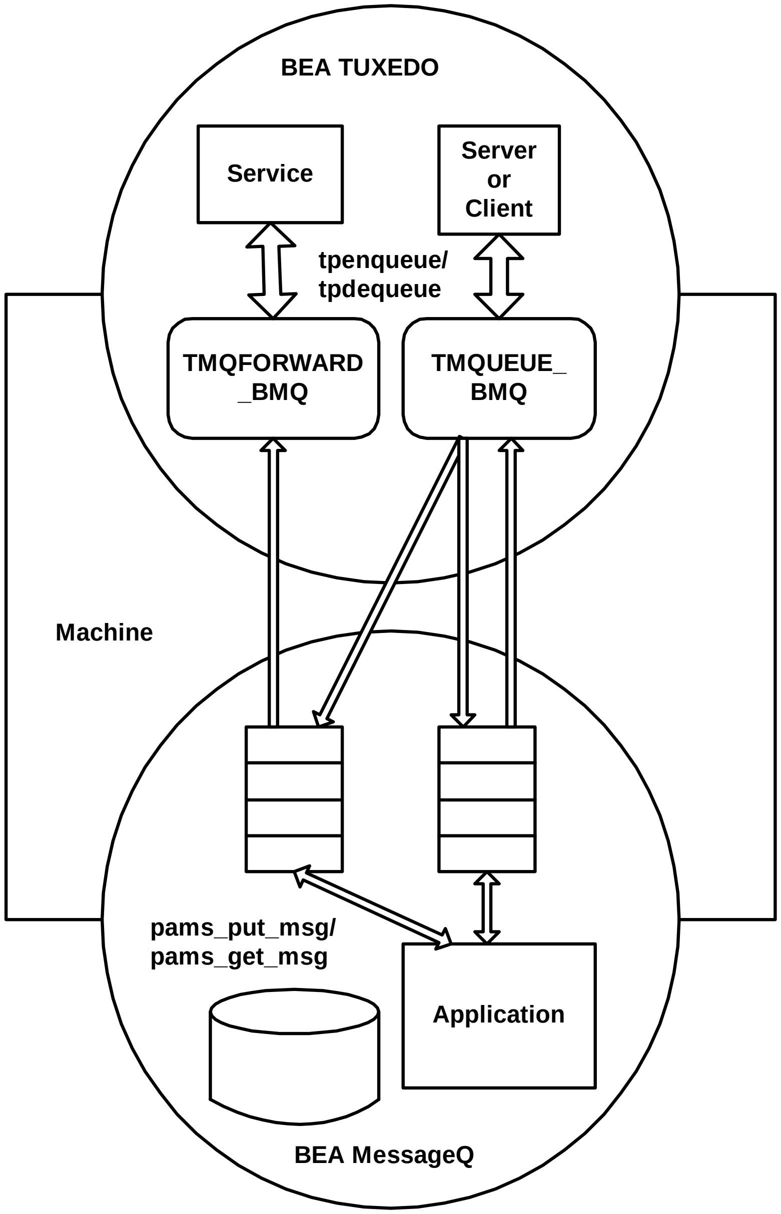
TMQUEUE_BMQ redirects TUXEDO `tpenqueue` requests to a BEA MessageQ queue where they can be retrieved with `pams_get_msg(w)`. TMQUEUE_BMQ also redirects `pams_put_msg` or `tepenqueue` requests to TUXEDO where they can be retrieved with `tpdequeue`.

TMQFORWARD_BMQ listens on specified BEA MessageQ queues and forwards `pams_put_msg` requests to a TUXEDO service. It also puts a reply or failure message on the sender's response queue.

The target queue and service are defined when TMQUEUE_BMQ and TMQFORWARD_BMQ are configured. This ensures that message exchange between BEA MessageQ and TUXEDO is transparent to the application.

Figure 1-2 illustrates message exchange between MessageQ and TUXEDO.

Figure 1-2 Message Exchange Between MessageQ and TUXEDO


BEA TUXEDO
Service
Server or Client
tpenqueue/
tpdequeue
TMQFORWARD _BMQ
TMQUEUE_ BMQ
Machine
pams_put_msg/
pams_get_msg
Application
BEA MessageQ

Enabling the Messaging Bridge

The TMQUEUE_BMQ and TMQFORWARD_BMQ servers are part of the BEA MessageQ kit and are installed when BEA MessageQ is installed. During the installation procedure, you are prompted to choose one of the following installation options for BEA MessageQ and TUXEDO integration:

install on top of BEA TUXEDO V6.4
install on top of BEA M3 2.1
install without BEA TUXEDO

If you choose to install on top of BEA TUXEDO V6.4 or BEA M3 2.1, the applicable files for the TMQUEUE_BMQ and TMQFORWARD_BMQ servers are installed on your system. If you install without BEA TUXEDO, the TMQUEUE_BMQ and TMQFORWARD_BMQ servers are not installed on your system. See the installation and configuration documentation specific to your platform for detailed installation and configuration instructions.

Once the TMQUEUE_BMQ and TMQFORWARD_BMQ servers are installed, the system administrator enables message enqueuing and dequeuing for the application by specifying the servers as application servers in the *SERVERS section of the TUXEDO `ubbconfig` file. See the TMQUEUE_BMQ and TMQFORWARD_BMQ reference pages in the *BEA MessageQ Reference Manual* for detailed information on the server configuration syntax.

Data Transformation Between BEA MessageQ and TUXEDO

One of the primary functions of the TMQUEUE_BMQ and TMQFORWARD_BMQ servers is to perform data and semantic transformations between the BEA MessageQ PAMS API and the TUXEDO ATMI API. This section describes how data is handled when it is exchanged between BEA MessageQ and TUXEDO. The data transformations are the same for the TMQUEUE_BMQ and TMQFORWARD_BMQ servers.

Data Types

BEA MessageQ passes data as static buffers or as FML32 buffers using the `msg_area` argument of the `pams_put_msg` function. TUXEDO handles a wide range of data types including `CARRAY`, `STRING`, and `FML32` using the `data` argument of the `tpenqueue` function.

When a message is enqueued using `tpenqueue`, the TMQUEUE_BMQ server preserves TUXEDO data type information for use by a subsequent call by `tpdequeue`. If machines of different types perform the `tpenqueue` and `tpdequeue` calls, and the data type is not `FML32` or `CARRAY`, the data is transformed to `CARRAY` and a message is written to the TUXEDO user log. (Machine types are specified in the TUXEDO `ubbconfig` file in the *MACHINE section using the TYPE attribute.)

When a message is enqueued using `pams_put_msg` and dequeued with `tpdequeue`, static buffer data is transformed to `CARRAY`, and `FML32` buffers are passed without transformation.

When a message is dequeued using `pams_get_msg(w)`, FML32 buffers are passed without transformation and all other data types are transformed to binary large objects.

Data Size and Length

BEA MessageQ defines the size and length of messages using the following arguments to `pams_put_msg`: `msg_size`, `large_size`, `msg_area_len`, `len_data`, and `large_area_len`. TUXEDO uses the `len` argument to `tpenqueue` to determine length.

BEA MessageQ limits the size of messages to a maximum of 4 MB. In addition, BEA MessageQ can be configured to set a smaller maximum message size. If BEA MessageQ is configured for a 4 MB maximum size, and a message larger than 4 MB is enqueued using `tpenqueue`, a `TPEDIAGNOSTIC`/`QMESYSTEM` error is generated. If BEA MessageQ is configured for a smaller maximum message size, and a message larger than the configured size is enqueued using `tpenqueue`, there is no way to detect the message size error.

When messages are dequeued using `tpdequeue`, the TMQUEUE_BMQ server handles buffer size discrepancies and returns a full, complete buffer to the calling application.

Timeouts

BEA MessageQ specifies a timeout per operation using the `timeout` argument of the `pams_put_msg` function. TUXEDO specifies system-wide blocking timeouts using the following flags: `ctl.flags:TPQWAIT`, `flags:TPNOBLOCK`, and `flags:TPNOTIME`.

When the TMQUEUE_BMQ server handles a message from a BEA MessageQ queue based on a call to `tpenqueue` or `tpdequeue`, the timeout is the value set by the TMQUEUE_BMQ command line option `-t`, or the default timeout if none is specified.

When a message is enqueued using `pams_put_msg` and is intended for a TUXEDO application, the timeout is the value set by `timeout` argument of the `pams_put_msg` function, within any limitations set by the BEA MessageQ delivery mode.

Priorities

BEA MessageQ specifies priority using the `priority` argument to the `pams_put_msg` function. TUXEDO specifies priority using the `ctl.flags:TPQPRIORITY` and `ctl.priority` flags. BEA MessageQ message priorities range from 0 to 99 with 99 being the highest priority. TUXEDO priorities range from 1 to 100 with 100 being the highest priority and the default being 50. BEA MessageQ requires that the `priority` argument of the `pams_put_msg` function be specified when the message is enqueued. TUXEDO uses the default priority if the control structure flag `ctl.flags:TPQPRIORITY` is not set.

Message priorities are either increased or decreased by one depending on where the message originates. Messages originating from TUXEDO are placed on the BEA MessageQ queue with a priority of *n*-1 where *n* is the priority assigned by TUXEDO. Messages originating from BEA MessageQ will dequeued by TUXEDO with a priority of *n*+1, where *n* is the priority assigned by BEA MessageQ.

Target, Queue Space and Queue Name

There are two areas that must be resolved when mapping the BEA MessageQ target and TUXEDO queue space and queue name:

- TUXEDO queue space to BEA MessageQ group name
- TUXEDO queue to BEA MessageQ queue

TUXEDO Queue Space to BEA MessageQ Group Name

BEA MessageQ uses the `target` argument of the `pams_put_msg` function to specify the target queue address for a message. TUXEDO uses the `qspace` and `qname` arguments of the `tpenqueue` and `tpdequeue` functions to specify the target queue for a message

The TUXEDO queue space name must be the name of a service advertised by TMQUEUE_BMQ or TMQFORWARD_BMQ. The service name maps directly to a BEA MessageQ group. By default, TMQUEUE_BMQ and TMQFORWARD_BMQ automatically offer services named "TMQUEUE_BMQ" and "TMQFORWARD_BMQ" unless the `-s` command line option is specified. These default services map to the BEA MessageQ group to which they are attached, as specified by the `-g` command line option.

The function name to which services should be mapped in TMQUEUE. Each entry in the TUXEDO `ubbconfig` file for a TMQUEUE_BMQ or TMQFORWARD_BMQ server should be configured with a different alias for the default function name using the TUXEDO `-s` command line option. For example, one configuration of TMQUEUE may be named Payroll, while another is named Sales. This provides a way to precisely specify a BEA MessageQ entry point for a particular `tpenqueue` or `tpdequeue` call. If multiple instances of the same advertised service are running, TUXEDO performs load balancing and data dependent routing to determine which server handles the request.

The following example illustrates different TMQUEUE_BMQ configurations:

```
*GROUPS
TMQUEUE_BMQGRPHQMGR GRPNO=1
TMQUEUE_BMQGRPHQPLEBE GRPNO=2
TMQUEUE_BMQGRPREMOTENA GRPNO=3
TMQUEUE_BMQGRPREMOTEEUROPE GRPNO=4


*SERVERS
TMQUEUE_BMQ SRVGRP="TMQUEUE_BMQGRPHQMGR" SRVID=1000 RESTART=Y
   GRACE=0 CLOPT="-s Payroll:TMQUEUE -s
   Promote:TMQUEUE -- -b 5 -g 7"
TMQUEUE_BMQ SRVGRP="TMQUEUE_BMQGRPHQPLEBE" SRVID=1000 RESTART=Y
   GRACE=0 CLOPT="-s Payroll:TMQUEUE -s
   Promote:TMQUEUE -- -b 5 -g 10"
TMQUEUE_BMQ SRVGRP="TMQUEUE_BMQGRPREMOTENA" SRVID=2002 RESTART=Y
   GRACE=0 CLOPT="-s Sales:TMQUEUE -- -b 5 -g 42"
TMQUEUE_BMQ SRVGRP="TMQUEUE_BMQGRPREMOTEEUROPE" SRVID=2002
```

```
   RESTART=Y GRACE=0 CLOPT="-s Sales:TMQUEUE -- -b 12 -g 53"


*SERVICES
Payroll  ROUTING="SALARYROUTE"
Payroll  ROUTING="HAIRCOLORROUTE"


*ROUTING
SALARYROUTE  FIELD=Salary BUFTYPE="FML32"
   RANGES="MIN - 50000:TMQUEUE_BMQGRPPLEBE,50001
   -MAX:TMQUEUE_BMQGRPHQMGR"
HAIRCOLORROUTE  FIELD=Hair BUFTYPE="FML32"
   RANGES="'Gray':TMQUEUE_BMQGRPHQMGR,*:TMQUEUE_BMQGRPPLEBE"
```

In this example, three queue space names (Payroll, Promote, and Sales) are defined for two busses to four different BEA MessageQ groups (7, 10, 42, and 53). Two servers offer the same aliases (Payroll and Promote) with data dependent routing performed using the Sales and Hair fields respectively. The two other servers offer the same alias (Sales) with routing determined by load balancing and availability.

TUXEDO Queue to BEA MessageQ Queue

Any BEA MessageQ queue can be accessed by TUXEDO through the TMQUEUE_BMQ and TMQFORWARD_BMQ servers. However, BEA MessageQ queues are accessed in different ways depending on whether they are named or unnamed queues. (For more information on BEA MessageQ naming capabilities, see Chapter 4, “Using Naming”.)

BEA MessageQ named queues can be local (group-wide) or global (bus-wide). To address a locally named queue from TUXEDO:

1. Configure the TMQUEUE_BMQ or TMQFORWARD_BMQ server to attach to the local group in which the named queue is defined.
2. Configure routing information to handle multiple instances of the TMQUEUE_BMQ or TMQFORWARD_BMQ server with the same alias as shown in “TUXEDO Queue Space to BEA MessageQ Group Name” on page 1-24.
3. Use the queue name as defined by BEA MessageQ as the second parameter for `tpenqueue` or `tpdequeue`.

To access an unnamed BEA MessageQ queue from TUXEDO, use an absolute queue identifier as the second parameter for `tpenqueue` or `tpdequeue`. The absolute queue identifier is a combination of the BEA MessageQ group identifier and queue identifier formatted as *group_id.queue_id*. For example, queue 1005 in group 3 is specified as "3.1005". When accessing a queue in the local group, either specify the group as 0 or drop the group identifier and delimiter. For example, queue 1005 in the local group is specified either as "0.1005" or "1005". Queue identifiers that do not use this syntax, or are outside the valid range of group or queue numbers are assumed to be queue names.

Delivery

When a message is enqueued using `tpenqueue`, the TMQUEUE_BMQ server uses the BEA MessageQ delivery mode of `PDEL_MODE_WF_SAF` (block until the message is stored in the local recovery journal). The exception to this occurs when the target queue is a temporary queue; in this case, the delivery mode `PDEL_MODE_WF_MEM` (block until message is stored in the target queue) is used.

If a confirmation delivery mode is required by the BEA MessageQ application, the queues attached to the TMQUEUE_BMQ server must be configured for explicit confirmation.

Messages handled by the TMQUEUE_BMQ server are recoverable, and message recovery services (MRS) must be enabled for the BEA MessageQ group. If MRS is not enabled, the attempt to enqueue the message will fail unless it is enqueued to a temporary queue where recoverable messaging is not required.

Undeliverable Messages

BEA MessageQ specifies the disposition of undeliverable messages according to an undeliverable message action (UMA). TUXEDO uses the `ctl.flags:TPQFAILUREQ` and `ctl.failurequeue` to specify a failure queue.

If a message is enqueued using `tpenqueue` and the `ctl.flags:TPQFAILUREQ` flag is set, the message is sent to BEA MessageQ with a UMA of `PDEL_UMA_DJL` (dead letter journal). If the target queue is a temporary queue, a UMA of `PDEL_UMA_DLQ` (dead letter queue) is used. The failure queue specified by `ctl.flags:TPQFAILUREQ` is preserved for use by `tpdequeue`. When BEA MessageQ dequeues a message enqueued by tpenqueue, the value of `ctl.failurequeue` is ignored.

When a TUXEDO application dequeues a message that was enqueued using tpenqueue, the value of `ctl.failurequeue` is returned to the application so that failure messages can be put on the failure queue. Failure queue names should be unique to avoid directing a failure message to the wrong queue.

Correlation Identifiers

BEA MessageQ and TUXEDO both support optional correlation identifiers stored as 32 character strings. No transformation is performed on either BEA MessageQ or TUXEDO correlation identifiers. When a response message is sent, the correlation identifier must be manually set.

Return Values

BEA MessageQ return values can be mapped to the TUXEDO `tperrno` and `ctl.diagnostic` values. The following table show the relationship between return values for calls to `tpenqueue`.

Table 1-1 Return Values for tpenqueue

MessageQ Return Value	TUXEDO tpperrno (return value = -1)	TUXEDO ctl.diagnostic
PAMS__BADPARAM	TPEDIAGNOSTIC	QMESYSTEM
PAMS__BADPRIORITY	TPEDIAGNOSTIC	QMESYSTEM
PAMS__BADPROCNUM	TPEDIAGNOSTIC	QMEBADQUEUE
PAMS__BADRESPQ	TPEDIAGNOSTIC	QMEBADQUEUE
PAMS__EXCEEDQUOTA	TPEDIAGNOSTIC	QMESYSTEM
PAMS__MSGTOBIG	TPEDIAGNOSTIC	QMENOSPACE
PAMS__NOTACTIVE	TPEDIAGNOSTIC	QMESYSTEM
PAMS__REMQFAIL	TPEDIAGNOSTIC	QMESYSTEM
PAMS__STOPPED	TPEDIAGNOSTIC	QMESYSTEM
PAMS__SUCCESS	N/A, return value = 0	
PAMS__TIMEOUT	TPEDIAGNOSTIC	QMESYSTEM

MessageQ Return Value	TUXEDO tpperrno (return value = -1)	TUXEDO ctl.diagnostic
`PAMS__UNATTACHEDQ`	N/A, return value = 0	
`PAMS__DLJ_FAILED`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__DLJ_SUCCESS`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__NO_UMA`	`TPEDIAGNOSTIC`	`QMESYSTEM`

The following table show the relationship between return values for calls to `tpdequeue`.

Table 1-2 Return Values for tpdequeue

MessageQ Return Value	TUXEDO tpperrno (return value = -1)	TUXEDO ctl.diagnostic
`PAMS__BADPRIORITY`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__INSQUEFAIL`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__MSGUNDEL`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__NETERROR`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__NOACCESS`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__NOACL`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__NOMOREMSG`	`TPEDIAGNOSTIC`	`QMENOMSG`
`PAMS__NOMRQRESRC`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__NOTDCL`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__PAMSDOWN`	`TPEDIAGNOSTIC`	`QMENOTOPEN`
`PAMS__REMQFAIL`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__STOPPED`	`TPEDIAGNOSTIC`	`QMESYSTEM`
`PAMS__SUCCESS`	N/A, return value = 0	

Other BEA MessageQ API Elements

The following arguments to BEA MessageQ PAMS API functions do not require a direct mapping to TUXEDO.

- `class` (`pams_put_msg`)—not accessible to TUXEDO applications. However, the TMQUEUE_BMQ server sets a class of `MSG_CLAS_TUXEDO` for messages generated by TUXEDO applications. Reply messages from BEA TUXEDO have either the BEA MessageQ class of `MSG_CLAS_TUXEDO_TPSUCCESS` or `MSG_CLAS_TUXEDO_TPFAIL`
- `type` (`pams_put_msg`)—not accessible to TUXEDO applications. The TMQUEUE_BMQ server does not return a type code (the value is NULL) for messages generated by TUXEDO applications.
- `psb` (`pams_put_msg`)—not accessible to TUXEDO applications.
- `resp_q` (`pams_put_msg`)—when BEA MessageQ specifies a response queue, the TMQUEUE_BMQ server uses that queue for responses from TUXEDO applications.
- `source` (`pams_get_msg`)—not accessible to TUXEDO applications.
- `sel_filter` (`pams_get_msg`)—not accessible to TUXEDO applications.
- `show_buffer` and `show_buffer_len` (`pams_get_msg`)—not accessible to TUXEDO applications.

Other TUXEDO API Elements

The following arguments to TUXEDO ATMI API functions do not require a direct mapping to BEA MessageQ.

- `ctl.flags:TPNOFLAGS`—no implications for TMQUEUE_BMQ.
- `ctl.flags:TPQTOP and ctl.flags:TPQBEFOREMSGID`—since BEA MessageQ orders queues by priority then FIFO order, if either of these flags is set in a control structure, a TPEINVAL error is generated and the error is logged in the TUXEDO user log.
- `ctl.flags:TPTIME_ABS`, `ctl.flags:TPQTIME_REL` and `ctl.deq_time`—since BEA MessageQ does not handle message generation time, if either of these flags is set in a control structure, a TPEINVAL error is generated and the error is logged in the TUXEDO user log.

- `ctl.flags:TPQREPLYQ` and `ctl.replyqueue`—any queue may be specified. If set, replies are directed to the specified queue. Queue names should be unique to avoid directing a reply message to the wrong queue.
- `ctl.flags:TPQMSGID`, `ctl.flags:TPQGETBEMSGID`, and `ctl.msgid`—the TUXEDO `msgid` specified in a `tpenqueue` control structure is preserved for use by a subsequent call to `tpdequeue`.
- `ctl.urcode`—the TUXEDO `urcode` specified in a `tpenqueue` control structure is preserved for use by a subsequent call to `tpdequeue`.
- `ctl.appkey and ctl.cltid`—these parameters are set to the identity assigned to the TMQUEUE_BMQ or TMQFORWARD_BMQ server receiving messages from BEA MessageQ; the original values are not preserved.
- `flags:TPNOTRAN`—if TMQUEUE_BMQ is requested from a transaction, and the TPNOTRAN flag is not set, a TPETRAN error is generated and the error is logged in the TUXEDO user log.
- `flags:TPSIGRSTRT`—no implications for TMQUEUE_BMQ.
- `flags:TPNOCHANGE`—the TMQUEUE_BMQ handles this flag as it would in TUXEDO. If the next data to be dequeued does not match the specified data type, the data is not dequeued and an error is generated.

CHAPTER

2 Using Recoverable Messaging

Applications send messages using the BEA MessageQ `pams_put_msg` function and one of two types of delivery modes: **recoverable** or **nonrecoverable**. If a message is sent as nonrecoverable, the message is lost if it cannot be delivered to the target queue unless the application incorporates an error recovery procedure. If the message is sent as recoverable, BEA MessageQ Message Recovery Services (MRS) automatically guarantee delivery to the target queue in spite of system, process, and network failures.

To ensure guaranteed delivery, the BEA MessageQ message recovery system writes recoverable messages to nonvolatile storage on the sender or receiver system. Then, if a message cannot be delivered due to an error condition, the message recovery system attempts redelivery of the message by reading it from the recovery journal until delivery is confirmed.

Application developers determine which messages should be sent as recoverable depending upon the needs of the application. Because recoverable messaging requires the extra step of storing the messages on disk, it requires additional processing time and power. To maximize performance, recoverable messaging should only be used when it is critical to application processing.

The BEA MessageQ message recovery system offers the following benefits:

- Reduces development time by eliminating the need for designing applications to recover messages that cannot be delivered.
- Prevents applications from losing data when applications, systems, or network links fail.
- Simplifies the implementation of an event-driven store and forward capability in networked applications.

BEA MessageQ also offers error recovery features for nonrecoverable messages such as the dead letter queue and the ability to return a message to the sender if the message cannot be delivered. This topic describes all of the BEA MessageQ delivery modes to enable you to understand the right choice for your application.

The following sections describe:

- Choosing a Message Delivery Mode
- How to Send a Recoverable Message
- How to Receive a Recoverable Message
- Using UMAs for Exception Processing

Recoverable Messaging on BEA MessageQ Clients

Choosing a Message Delivery Mode

The choice between recoverable and nonrecoverable delivery is based upon the needs of the application. Nonrecoverable messaging is used by applications that will not fail if some data is lost. For example, an application that continuously monitors and reports temperature readings every second would not use recoverable messaging. If one message is lost, the next message will arrive in one second.

However, some applications require that messages be delivered in spite of system, process, and network failures. For example, a shop-floor monitoring system may continuously collect information from supervisory control applications connected to production lines. This information is sent using nonrecoverable messaging to the monitoring application on the same system.

At the end of the shift, totals are accumulated and sent to the Manufacturing Resource Planning (MRP) system on the corporate mainframe to update inventory control and other applications. The shift totals are sent as a recoverable message to ensure that the MRP system is properly updated daily or that the appropriate error handling takes place. The application uses the BEA MessageQ message recovery system to guarantee message delivery without application intervention.

To determine the appropriate method for sending a message, the application developer decides:

- Does the application need to know if the message arrived at the target queue?
- If notification is required, how far must the message get before the sender program receives notification that the message has arrived?
- Should the application wait for notification or should it continue processing and receive notification through an asynchronous acknowledgment message?
- If the message is designated as recoverable, does the application need to know if the message has been stored by the recovery system?
- If the message is designated as recoverable, what should happen if it cannot be stored by the message recovery system?

The `delivery mode` argument of the `pams_put_msg` function determines:

- Whether the message is sent as recoverable or nonrecoverable
- Whether a blocking or nonblocking mode is selected
- Whether the sender program receives notification and how it is received
- The point in the message flow at which the notification is sent

The following sections describe:

- How the Message Recovery System Works
- Choosing Recoverable and Nonrecoverable Delivery Modes
- Choosing an Undeliverable Message Action

How the Message Recovery System Works

When an application sends a message across a communications network, the final receipt of the message can be interrupted by a variety of failure conditions. When a recoverable delivery option is used to send a message, BEA MessageQ software stores the message on a disk until the message is successfully delivered.

BEA MessageQ uses message recovery journals to store messages that are designated as recoverable. The message recovery journal on the local system is called the **store and forward (SAF)** file. The message recovery journal on the remote system is called the **destination queue file (DQF)**. If a recoverable message cannot be delivered, it is stored in either the SAF or DQF file and is automatically re-sent once communication with the target group is restored.

BEA MessageQ uses auxiliary journal files to provide additional message recovery capabilities. The **dead letter journal (DLJ)** file provides disk storage for messages that could not be stored for automatic recovery by the message recovery system. Undelivered messages stored in the DLJ file can be re-sent under user or application control.

The **postconfirmation journal (PCJ)** file stores successfully confirmed recoverable messages. It forms an audit trail of message exchange that can be read or printed. The PCJ file can also be used to resend successfully delivered messages if a database has become corrupted and must be restored. The message queuing group must be configured to store successfully delivered messages in the PCJ file.

If the BEA MessageQ message recovery system is unable to store the message, the **undeliverable message action (UMA)** is taken. Some UMAs enable the message to be recovered at a later time under user or application control.

Choosing Recoverable and Nonrecoverable Delivery Modes

The delivery mode is specified as a constant consisting of two components, the **sender notification code (sn)** and the **delivery interest point (dip)**, as follows:

`PDEL_MODE_`*`sn_dip`*

where:

- *`sn`*—indicates how the sender program wants to receive information about the delivery of the message. You can wait for the operation to complete (WF), receive notification in an asynchronous message (AK), or choose not to receive notification (NN).

- *`dip`*—determines whether the message is designated as recoverable. When the message reaches the delivery interest point, a notification message is sent (if requested) and the call returns control to the sender program or BEA MessageQ delivers the asynchronous acknowledgment message.

Nonrecoverable delivery interest points enable the sender program to receive notification when the message is stored in the target queue (MEM), when the message is read from the target queue (DEQ), or when the message is read from the target queue and explicitly confirmed by the receiver program using the `pams_confirm_msg` function (ACK).

When a recoverable delivery interest point is selected, the message is stored on disk for automatic recovery. Recoverable delivery interest points enable the sender program to store the message in the local recovery journal (SAF), store the message in the remote recovery journal (DQF), or store the message in the remote recovery journal and receive notification when the message is confirmed by the target application (CONF).

BEA MessageQ does not support all possible combinations of sender notification code and delivery interest points. Table 2-1 describes all of the valid BEA MessageQ delivery modes and their meanings.

Table 2-1 Supported Delivery Modes

Delivery Mode	**Description**
(Recoverable Delivery Modes)	
`PDEL_MODE_AK_CONF`	Send acknowledgment message when the message recovery system confirms message delivery from the remote recovery journal.
`PDEL_MODE_AK_DQF`	Send acknowledgment message when the message is stored in the remote recovery journal.
`PDEL_MODE_AK_SAF`	Send acknowledgment message when the message is stored in the local recovery journal.
`PDEL_MODE_NN_DQF`	Deliver message to the remote recovery journal but do not block and do not send notification.
`PDEL_MODE_NN_SAF`	Deliver message to the local recovery journal but do not block and do not send notification.
`PDEL_MODE_WF_CONF`	Block until the message is stored in the remote recovery journal and confirmed by the target application.

Table 2-1 Supported Delivery Modes

Delivery Mode	Description
`PDEL_MODE_WF_DQF`	Block until the message is stored in the remote recovery journal.
`PDEL_MODE_WF_SAF`	Block until the message is stored in the local recovery journal.
(Nonrecoverable Delivery Modes)	
`PDEL_MODE_AK_ACK`	Send acknowledgment message when the receiver program explicitly confirms delivery using `pams_confirm_msg`.
`PDEL_MODE_AK_DEQ`	Send acknowledgment message when the message is removed from the target queue.
`PDEL_MODE_AK_MEM`	Send acknowledgment message when the message is stored in the target queue.
`PDEL_MODE_NN_MEM`	Deliver message to the target queue but do not block and do not send notification.
`PDEL_MODE_WF_ACK`	Block until the receiver program explicitly confirms delivery using `pams_confirm_msg`.
`PDEL_MODE_WF_DEQ`	Block until the message is removed from the target queue.
`PDEL_MODE_WF_MEM`	Block until the message is stored in the target queue.

The following sections describe:

- When to Use Nonrecoverable Message Delivery
- When to Use Recoverable Message Delivery

When to Use Nonrecoverable Message Delivery

Nonrecoverable message delivery is the fastest and most efficient way to send messages. Use nonrecoverable delivery modes if:

- High messaging rates are required by the application (hundreds or thousands of messages per second).

- The message content has a finite lifetime; therefore, the value of the information is stale if not received and processed quickly.
- The message is sent locally between two applications in the same message queuing group that tightly cooperate in the processing of an event.
- The message is a control message that causes a component of an application to change state.

When to Use Recoverable Message Delivery

Recoverable message delivery is the safest way to send a message; however, it adds significant processing overhead because each message must be stored on disk before it is sent. Use recoverable delivery modes if:

- It is useful to know that the message has arrived; however, the sender does not need to know the state of the receiver.
- The message content should not be lost by the application system.
- The application can tolerate the increased system load and slower messaging rate caused by sending the message recoverably.

Choosing an Undeliverable Message Action

Using the `pams_put_msg` function in conjunction with the `delivery` argument, you can use the `uma` argument to specify what should happen to the message if it cannot be delivered to the delivery interest point. For nonrecoverable messaging, if a UMA is not specified, BEA MessageQ will take the default action of discarding the message.

With recoverable messaging, the UMA indicates the action to be taken if the message cannot be stored in either the SAF or DQF files. You must specify a UMA with recoverable delivery modes because your application must perform the exception processing when the message cannot be guaranteed for delivery by BEA MessageQ.

With recoverable messaging, the UMA may be taken when:

- The message recovery system journal process on the local or target node is not running.

- BEA MessageQ is unable to write to the local journal disk file (SAF) where the message is designated to be recoverable.
- The cross-group connection to the remote target group is down and the message designated as recoverable on the remote node (DQF) cannot be stored.
- The system resources used by the message recovery system are exhausted.
- On OpenVMS systems, if the system manager has disabled message recovery for a particular queue.

Table 2-2 lists the six valid UMAs.

Table 2-2 Valid UMAs

UMA	Description
`DISC`	Discard—the message is deleted.
`DISCL`	Discard and log—the message is deleted and an entry indicating that the message was not stored by the message recovery system is added to the BEA MessageQ event log. `DISCL` is available on OpenVMS only. Though you can specify the `DISCL` UMA on UNIX and Windows NT systems, it discards the message without logging the event.
`RTS`	Return to sender—the message is delivered to the sender's response queue.
`DLQ`	Dead letter queue—the message is written to the dead letter queue. This queue is permanently active queue number 96, called the `PAMS_DEAD_LETTER_QUEUE`.
`DLJ`	Dead letter journal—the message is written to the DLJ file. From the DLJ file, the message can be re-sent at a later time under user or application control.
`SAF`	Store and forward—the message is written to the message recovery journal on the sender system.

See the Using UMAs for Exception Processing topic for a description of how to use each UMA for exception handling with recoverable messaging.

How to Send a Recoverable Message

To send a recoverable message, use the `pams_put_msg` function supplying the appropriate `delivery` and `uma` arguments. In addition, the application should:

- Specify a timeout value—applications can adjust the timeout value when sending recoverable messages with blocking delivery modes. The timeout value is adjusted to suit system loads.
- Check the delivery outcome—applications should always verify the delivery outcome of a send operation to know what happened to the message. If the message was not stored by the message recovery system, the application must check to make sure that the UMA was successfully executed.

The message flow for sending a recoverable message is:

1. The application sends a message using the `pams_put_msg` function and the appropriate `delivery` and `uma` arguments.
2. The message recovery system returns a sequence number to the sender program.
3. The message recovery system writes the message to the recovery journal on the local or remote system depending upon the delivery mode specified.
4. The sender program is notified that the message is stored on disk.
5. If the sender program is blocked, it continues processing once the message is received at the delivery interest point. If the sender program requested notification, it receives an acknowledgment message once the message reaches the delivery interest point.

Sequence Numbers

Sequence numbers are unique across all applications and across all groups within a single message bus. Ordering by sequence number only has meaning in relation to the sending application. For example, if two applications send messages to a queue, there is no guarantee that application A has higher or lower sequence numbers than application B. In addition, it is possible for sequence numbers to wrap, causing a new message to have a lower sequence number than an older message.

Sequence numbers are composed of the following:

- a time in seconds since January 1, 1970
- one bit (an extra bit used to extend a counter)
- a group number
- a 16-bit counter within the second

If the application sends 65536 messages within a single second, the extra bit is used to make the counter a 17-bit number.

Any single application generates monotonically increasing serial numbers. You cannot count on monotonically increasing serial numbers across multiple applications (this includes both local and cross group communications). This is especially true of cross group communications since the sequence number contains the originating group number.

Specifying Timeout Values

A timeout argument can be supplied to the `pams_put_msg` function to prevent the sender program from blocking indefinitely while waiting for the message recovery system to store a message. If the timeout expires before the message is stored in the SAF or DQF, BEA MessageQ returns control to the sender program and returns the `PAMS__TIMEOUT` status return.

When specifying a timeout with a send operation, it is important to provide ample time for the operation to complete successfully. For example, if the application normally delivers many messages each second, setting the timeout argument to 30 seconds should provide adequate time for the operation to complete.

Receiving a timeout return status represents a significant system failure. When a timeout occurs, either the message queuing load to the message recovery system is abnormally high or too much time is required to store the message due to disk I/O delays or CPU loading. The timeout return status cannot reflect whether the message was successfully stored by the message recovery system.

The sender program should include error handling routines for the `PAMS__TIMEOUT` status return. Receipt of a timeout return status indicates that messaging load and traffic should be examined as well as the MRS group configuration to ensure that all processes are configured and working properly.

Because the sender program cannot be sure whether the message was stored by the recovery system, the receiver program could receive duplicate messages if the message is re-sent. Therefore, using a timeout on the send operation may not be appropriate for applications that would experience processing problems if duplicate information is received.

Checking Delivery Outcome

There are several status return values that should be checked to verify the success or failure of the attempt to send a recoverable message:

- Return status of the `pams_put_msg` call
- The success or failure message indicates whether the message was successfully stored by the message recovery system.
- PAMS Status Block (PSB) returned by the `pams_put_msg` call or the `MRS_ACK` asynchronous acknowledgment message. The PSB is a BEA MessageQ data structure that delivers detailed status information about a send or receive operation.

Table 2-3 describes the fields in the PSB.

Table 2-3 PAMS Status Block

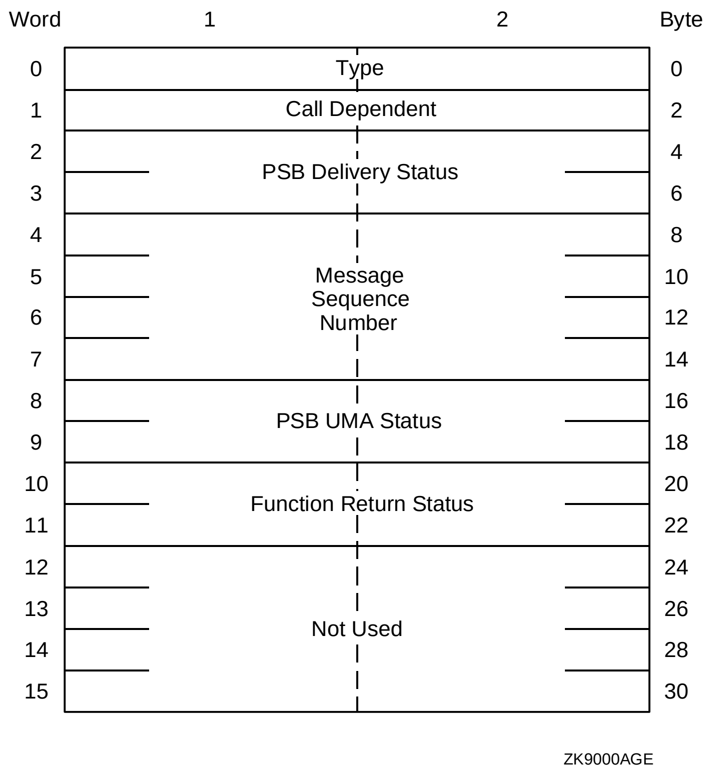
Field Name	Description
PSB Type	Type number of the PSB structure. BEA MessageQ Version 3.0 uses PSB structure type 2.
Call Dependent	Field not currently used.
Delivery Status	The completion status of the function call. It contains the status from the message recovery system. It can also contain a value of `PAMS__SUCCESS` when the message is not sent recoverably.
Message Sequence Number	A unique number assigned to a message when it is sent and follows the message to the destination PSB. This number is input to the `pams_confirm_msg` call to release a recoverable message.

Table 2-3 PAMS Status Block

Field Name	Description
PSB UMA Status	The completion status of the undeliverable message action (UMA). The PSB UMA status indicates whether the UMA was not executed or applicable.
Function Return Status	After a BEA MessageQ routine completes execution, BEA MessageQ software writes the return value to this field.

Figure 2-1 illustrates the size and location of the fields in the PSB.

Figure 2-1 PAMS Status Block



When an application sends a recoverable message, there are two ways to request notification that the recoverable message is delivered to the delivery interest point. The blocking approach (WF) causes the application to suspend processing until the `pams_put_msg` function is completed. Using WF notification, the `pams_put_msg` function returns all information required to determine the outcome of recoverable message delivery.

The other notification request method is asynchronous acknowledgment (AK), which enables the application to continue processing while the message is delivered to the delivery interest point. In this case, some status information is supplied by the `pams_put_msg` function and the balance is obtained using the `pams_get_msg` function to read the MRS acknowledgment message returned to the sender program's response queue.

The following sections describe:

- Checking the Delivery Status of WF Requests
- Checking the Delivery Status of AK Requests

Checking the Delivery Status of WF Requests

To determine the outcome of recoverable delivery using WF notification, follow these procedures:

If the return status of the `pams_put_msg` function is PAMS__SUCCESS, check the PSB delivery status to determine the outcome of the delivery. If this field contains a success status, the message has been successfully stored by the message recovery system. Extract the message sequence number from the PSB. Table 2-4 lists the valid PSB delivery status returns.

Table 2-4 PSB Delivery Status Values

PSB Delivery Status Returns	Status	Description
`PAMS__CONFIRMREQ`	Information	Confirmation required for this message.
`PAMS__DQF_DEVICE_FAIL`	Failure	Message not recoverable; destination queue file (DQF) I/O failed.
`PAMS__DQF_FULL`	Failure	Message not recoverable; DQF full.
`PAMS__ENQUEUED`	Success	Message is recoverable.

Table 2-4 PSB Delivery Status Values

PSB Delivery Status Returns	Status	Description
`PAMS__MRS_RES_EXH`	Failure	Message not recoverable; file system ran out of space or other resources, or incorrect configuration of the DQF or SAF.
`PAMS__NO_DQF`	Failure	Message not recoverable; no DQF for target queue.
`PAMS__POSSDUPL`	Information	Message is a possible duplicate.
`PAMS__SAF_DEVICE_FAIL`	Failure	Message not recoverable; store and forward (SAF) I/O failed.
`PAMS__SAF_FORCED`	Success	Message written to SAF file to maintain FIFO order.
`PAMS__STORED`	Success	Message is recoverable.
`PAMS__SUCCESS`	Success	Indicates successful completion.

If the PSB delivery status field contains a failure status, check the PSB UMA status to determine the outcome of the UMA. If the field contains a success status, the UMA was executed. If the UMA was not successfully executed, the message was lost and must be resent. Table 2-5 lists the PSB UMA status returns.

Table 2-5 UMA Status Values

UMA Status Returns	Status	Description
`PAMS__DISC_FAILED`	Failure	Message not recoverable in destination queue file (DQF); UMA was `PDEL_UMA_DISC`; message could not be discarded.
`PAMS__DISC_SUCCESS`	Success	Message not recoverable in DQF; UMA was `PDEL_UMA_DISC`; message discarded.
`PAMS__DISCL_FAILED`	Failure	Message not recoverable in DQF; UMA was `PDEL_UMA_DISC`; recoverability failure could not be logged or message could not be discarded.

Table 2-5 UMA Status Values

UMA Status Returns	Status	Description
`PAMS__DISCL_SUCCESS`	Success	Message not recoverable in DQF; UMA was `PDEL_UMA_DISCL`; message discarded after logging recoverability failure.
`PAMS__DLJ_FAILED`	Failure	Message not recoverable in DQF; UMA was `PDEL_UMA_DLJ`; dead letter journal file (DLJ) write operation failed.
`PAMS__DLJ_SUCCESS`	Success	Message not recoverable in DQF; UMA was `PDEL_UMA_DLJ`; message written to dead letter journal.
`PAMS__DLQ_FAILED`	Failure	Message not recoverable in DQF; UMA was `PDEL_UMA_DLQ`; message could not be queued to the dead letter queue.
`PAMS__DLQ_SUCCESS`	Success	Message not recoverable in DQF; UMA was `PDEL_UMA_DLQ`; message queued to the dead letter queue.
`PAMS__NO_UMA`	Success	Message is recoverable; UMA not executed.
`PAMS__RTS_FAIL`	Failure	Message not recoverable in DQF; UMA was `PDEL_UMA_RTS`; message could not be returned to sender.
`PAMS__RTS_SUCCESS`	Success	Message not recoverable in DQF; UMA was `PDEL_UMA_RTS`; message returned.
`PAMS__SAF_FAILED`	Failure	Message not recoverable in DQF; UMA was `PDEL_UMA_SAF`; store and forward (SAF) write operation failed.
`PAMS__SAF_SUCCESS`	Success	Message not recoverable in DQF; UMA was `PDEL_UMA_SAF`; message recoverable from SAF file.
`PAMS__UMA_NA`	Success	UMA not applicable.

Checking the Delivery Status of AK Requests

When a message sent with a `PDEL_MODE_AK` delivery mode reaches its delivery interest point, the message recovery system sends an `MRS_ACK` message back to the sender program using the queue name or number indicated in the `resp_q` argument. The sender program uses the `pams_get_msg` or `pams_get_msgw` functions to read the `MRS_ACK` message from its response queue.

The PSB returned by the `MRS_ACK` message contains the message sequence number of the previously sent recoverable message. The message sequence number of the `MRS_ACK` message is matched to the message sequence number returned by the `pams_put_msg` function before confirming message receipt. See Chapter 9, "Message Reference" for a detailed description of the `MRS_ACK` message format.

If temporary queues are used, deleted, and reused quickly, it is possible that an acknowledgment from an earlier instance of the queue can be retrieved on a later instance of the queue. Care should be taken when reusing temporary queues.

If the return status is `PAMS__SUCCESS`, the message has been successfully stored by the message recovery system. The message sequence number should be extracted from the PSB and saved until the acknowledgment message is received.

Follow these steps to determine the outcome of message delivery by reading the PSB returned in the `MRS_ACK` message:

1. Check the PSB delivery status. If this field contains a success status, the message is recoverable. The message sequence number should be extracted from the PSB and compared to the previously saved message sequence number. PSB Delivery Status Values lists the valid PSB delivery status returns and their meaning.
2. If the PSB delivery status contains a failure status, check the PSB UMA status to determine the outcome of the UMA. If the field contains a success status, the UMA was executed. If the UMA was not successfully executed, the message is lost and must be re-sent. UMA Status Values lists the PSB UMA status returns.

How to Receive a Recoverable Message

To receive a recoverable message, use the `pams_get_msg`, `pams_get_msgw`, or `pams_get_msga` functions. When a recoverable message is delivered to the target queue, the application must perform the following:

- Confirm message receipt—messages stored by the message recovery system must be deleted from the recovery journal once they are successfully delivered to the target queue. Message confirmation informs the message recovery system to delete the message to avoid sending duplicate messages if a failure condition causes the content of the recovery journal to be re-sent.

  In the group initialization file, message queues are configured to require either implicit or explicit confirmation. Explicit confirmation of recoverable messages requires the application to call the `pams_config_msg` function when a recoverable message is received. Implicit confirmation means that BEA MessageQ automatically confirms receipt of the recoverable message after it is dequeued and a subsequent dequeue operation has occurred on the target queue.

- Check for duplicate messages—applications may check for duplicate messages based on the type of task performed. For example, if a banking application did not check for duplicate transactions, duplicate deposits or withdrawals could be posted to a customer's account. On the other hand, a stock brokerage application that receives continuously updated stock prices would not be adversely affected by duplicate stock price quotations.

The message flow for receipt of a recoverable message by the target system is as follows:

1. A message is read from the message recovery journal by the recovery system and sent to the target queue of the receiver program.

2. The receiver program reads the `pams_get_msg`, `pams_get_msgw`, or `pams_get_msga` functions.

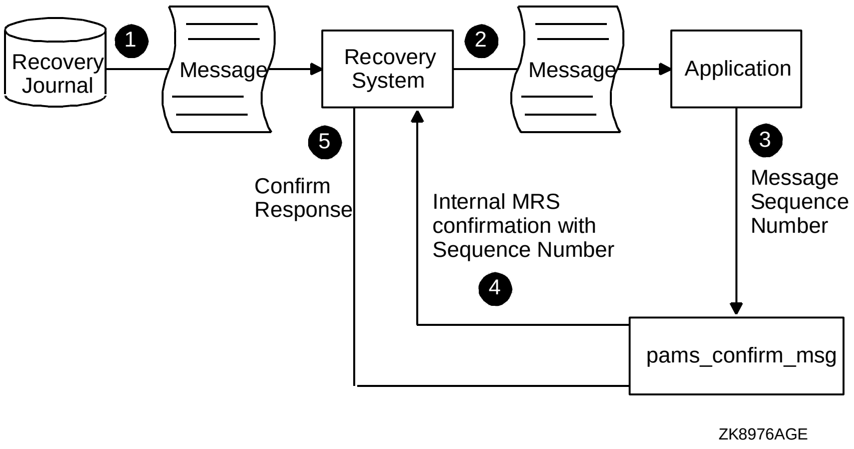
3. If the queue is configured for explicit confirmation, the application calls the `pams_confirm_msg` function to acknowledge receipt of the recoverable message using the message sequence number assigned by the message recovery system when the message was sent. If the queue is configured for implicit confirmation, BEA MessageQ performs this function after the recoverable message is delivered to the target queue.

4. The `pams_confirm_msg` function sends notification to the message recovery system that the message was delivered and awaits a response.

5. The message recovery system removes the message from the message recovery journal and sends a nonblocking message back to the `pams_confirm_msg` function.

Figure 2-2 illustrates the message flow for receiving a recoverable message.

Figure 2-2 Message Flow for Receiving a Recoverable Message



The following sections describe:

- Confirming Message Receipt
- Checking for Duplicate Messages

Confirming Message Receipt

When the receiver program reads a recoverable message from its target queue, the recovery system retains the message until delivery is confirmed. The `pams_confirm_msg` function is used to remove successfully delivered recoverable messages from the message recovery journal. The message recovery system attempts redelivery of recoverable messages from the recovery journal each time the target queue detaches from and reattaches to the message queuing bus.

The receiver program reads the PSB delivery status of each message to know which messages to confirm. A PSB delivery status of `PAMS__CONFIRMREQ` indicates that the message requires confirmation. A PSB delivery status of `PAMS__POSSDUPL` also requires confirmation to delete the message from the message recovery system.

The following sections describe:

- Selecting a Confirmation Type
- Selecting a Confirmation Order
- Creating an Audit Trail of Confirmed Messages

Selecting a Confirmation Type

BEA MessageQ offers the following two types of message confirmation:

- Implicit confirmation—enables the BEA MessageQ recovery system to automatically call the `pams_confirm_msg` function to delete a recoverable message. Implicit confirmation is triggered when the next sequential message for that queue is read from the journal file using the `pams_get_msg` call.
- Explicit confirmation—requires the receiver program to call the `pams_confirm_msg` function to delete the message from the message recovery journal. The `pams_confirm_msg` function uses the message sequence number supplied in the PSB when the user receives the message. The `pams_confirm_msg` function should not be called until the receiver program has completed processing the information in the message.

Implicit confirmation frees receiver programs from the need to respond to the receipt of a recoverable message. If you are using implicit confirmation with recoverable messaging, you must ensure that the last message is confirmed before detaching from the queue, exiting BEA MessageQ, or exiting your application. If the message is not properly confirmed, it will be redelivered when the queue is reattached.

Explicit confirmation is normally used when several messages are required to contain a single transaction or work unit. The application reads each message until all the data is present, applies the data, and then confirms all the messages involved in the transaction at once.

All queues must be configured for implicit or explicit confirmation. For complete information on how to configure message queues, see the *Installation and Configuration Guide* for the platform you are using.

Selecting a Confirmation Order

Confirmation order is another MRS configuration characteristic that can affect how recoverable messages are confirmed by the receiver program. Queues can be configured to confirm messages in order or out-of-order. The default configuration used for each message queuing group is to confirm messages in order.

If confirmation is in order, messages must be confirmed in the order in which they are received. If confirmation is out-of-order, then messages can be confirmed in any order. For more information on how to set confirmation order, see the installation and configuration guide for the platform you are using.

Creating an Audit Trail of Confirmed Messages

When using recoverable messaging, you can choose to write successfully delivered recoverable messages to the postconfirmation journal (PCJ) of the target group. The contents of the postconfirmation journal forms an audit trail of successfully delivered messages that you can print out or use to resend messages in the event of a database rollback.

To use PCJ journaling, you must do the following:

- Set the `ENABLE_JRN` parameter in the `%PROFILE` section of the group initialization file to `YES`. The default journaling action is not to write messages to the PCJ.
- Specify the path name for the PCJ file in the `%MRS` section of the initialization file. On OpenVMS systems, the file specification for the PCJ is automatically created when you enable MRS.
- Configure the target queue that will receive the messages to require explicit confirmation. If a queue is configured for implicit confirmation, no journaling of successfully delivered messages takes place regardless of whether journaling is enabled.
- Use the `pams_confirm_msg` function to explicitly confirm messages and set the `force_j` argument to `PDEL_FORCE_JRN` to store successfully delivered recoverable messages in the PCJ. To prevent messages from being stored in the PCJ when journaling is enabled, set the `force_j` argument to `PDEL_NO_JRN`. Note that if journaling is not enabled in the group initialization file, no messages are written to the PCJ file regardless of the value of the `force_j` argument.

For OpenVMS applications, you can also set the `force_j` argument to `PDEL_DEFAULT_JRN` to use the default journaling action. The default journaling action can be changed using the `MRS_SET_PCJ` message.

On UNIX and Windows NT systems, messages stored in the PCJ file can be re-sent using the `dmqjplay` utility and dumped using the `dmqdump` utility. For instructions on how to use the MRS utilities, see the installation and configuration guide for your platform

To read or resend journaled messages on BEA MessageQ for OpenVMS systems, use the `pams_open_jrn`, `pams_read_jrn`, and `pams_close_jrn` functions to open, read, and close the PCJ file. See the Application Programming Interface topic for a detailed description of these functions. For information on how to use MRS utilities to resend or dump the contents of the PCJ, see the *BEA MessageQ Configuration Guide for OpenVMS*.

On OpenVMS systems, the default journaling action can be set under program control by sending an `MRS_SET_PCJ` message to the MRS Server process. The current PCJ file can also be closed and a new one opened by the same message. Because UNIX and Windows NT do not currently support the `MRS_SET_PCJ` message, the default journaling action can not be changed. This mean that the only way to write messages on these systems is to specify a `force_j` value of `PDEL_FORCE_JRN`.

Checking for Duplicate Messages

If recoverable message delivery is not properly confirmed by the receiver program, duplicate messages can be delivered to the target application. For example, a message may be sent from a recovery journal, but the cross-group connection may be lost before the message confirmation is delivered.

When the cross-group connection is reestablished, the message will be resent from the message recovery journal and carry a PSB delivery status of `PAMS__POSSDUPL`. The receiver program must check for this PSB delivery status if the posting of duplicate information will cause processing errors.

The PSB delivery status `PAMS__POSSDUPL` does not always indicate a duplicate message. If receipt of a duplicate message will cause processing problems, the receiver program must include the logic to determine whether the message marked with the `PAMS__POSSDUPL` delivery status is indeed a duplicate of a message already received.

Using UMAs for Exception Processing

An undeliverable message action (UMA) must be specified for each recoverable message. The UMA provides the application developer with a variety of ways to perform exception handling when the message cannot be stored for guaranteed delivery by the message recovery system. Table 2-6 describes the UMAs supported by BEA MessageQ.

Table 2-6 How to Use UMAs

If you want to...	Use...	Description
Handle each exception immediately	`DISC`	The sender program is coded to handle each exception immediately with an application-specific response. The message is discarded by the messaging system because the application holds the message in memory and attempts recovery. The sender program sends each message and is responsible for handling all error recovery and redelivery of each message.
Handle each exception immediately and log errors	`DISCL`	This UMA is available only on OpenVMS systems. The sender program is coded to handle each exception immediately with an application-specific response. BEA MessageQ writes a description of the exception condition to the error log. The log can be used by system managers to track and diagnose system problems. The sender program sends each message, and is responsible for handling all error recovery, logging the error event, and redelivering each message.

Table 2-6 How to Use UMAs

If you want to...	Use...	Description
Handle errors by redirecting them to the sender program's input stream	`RTS`	The sender program directs undeliverable messages to its queue, eliminating the need to handle each error as it happens. Using the RTS UMA, the sender program uses the attachment to its primary queue to read new messages and handle error conditions for messages that could not be delivered. The sender program sends each message, and must check each message to see if it was returned or if it is a new message sent by another process. If the message was returned, the sender program is responsible for handling all error recovery and redelivering each message.
Handle errors by reading them from a central queue	`DLQ`	The sender program directs undeliverable messages to a special queue separate from the main input stream for the program that is designed to hold undeliverable messages. Using this approach, the application makes an additional attachment to the dead letter queue and handles each exception as it is read from the DLQ. Because the undeliverable messages are stored in a queue, they will be lost if the system goes down.
Handle all errors by reading them from a file	`DLJ`	The sender program directs all undeliverable messages to a file. Undeliverable messages can be re-sent from the DLJ under user or application control. Selection criteria can be applied enabling the user or application to attempt redelivery on a subset of messages. Because messages are stored in a file, they will not be lost if the system goes down and they can be re-sent until they can be delivered. The application must develop an additional process or system management procedures must be created to deliver messages from the DLJ.
Establish recoverability locally or remotely	`SAF`	Any message that cannot be delivered to the remote recovery journal is redirected for storage by the local recovery journal. Because the UMA may fail, you cannot guarantee that a message will be stored by the message recovery system.

To choose the appropriate error handling technique and corresponding UMA, the application developer must analyze the consequences to application processing if a message is not stored for guaranteed delivery. If a message is critical, it is best to perform exception processing immediately to attempt resolution of the failure condition. If the receipt of the message is not time-critical, centralized mechanisms such as DLQ and DLJ may be preferable. The Supported Delivery Modes and UMAs topic contains a complete list of the supported combinations of delivery modes and UMAs.

The following sections describe:

- Using Discard and Discard and Log UMAs
- Using the Return-to-Sender UMA
- Using the Dead Letter Queue UMA
- Using the Dead Letter Journal
- Using the SAF UMA

Using Discard and Discard and Log UMAs

When the `DISC` UMA is used, the message is discarded if it cannot be delivered to the delivery interest point specified in the delivery mode argument. The `DISC` UMA is used when the sender program will handle each exception as it occurs. BEA MessageQ can discard the undeliverable message because the message content is still available in the context of the sender program. To log the undeliverable message event, use the `DISCL` UMA.

Because the sender program cannot be sure that the UMA will be executed successfully, handling exceptions on a message-by-message basis is the safest way to ensure that the application recovers properly from error conditions. In addition, on OpenVMS systems, using the `DISCL` UMA creates an event log that can be used to track and diagnose system problems.

Note: On UNIX and Windows NT systems, the `DISCL` UMA functions the same as the `DISC` UMA.

Using the Return-to-Sender UMA

When the `RTS` UMA is used, the message is directed to the response queue of the sender program if it cannot be delivered to the delivery interest point specified in the delivery mode argument. The `RTS` UMA is used when the sender program does not want to process each exception as it occurs. Instead, the sender program redirects undeliverable messages to its main input stream for error handling.

The advantage to using the `RTS` UMA is that the sender program attaches to one queue and acts upon each message as it is read. The sender program must read the PBS status delivery value of each message to determine if the message is new or an undeliverable message. Messages that could not be stored by the message recovery system and require error handling have a return status of `PAMS__MSGUNDEL`.

Using the Dead Letter Queue UMA

When the `DLQ` UMA is used, the message is redirected to queue number 96 (the dead letter queue) if it cannot be delivered to the delivery interest point specified in the delivery mode argument. The `DLQ` UMA is used when the sender program wants to centralize error handling for undeliverable messages in a designated queue while allowing each message to be handled separately.

A dead letter queue is part of the standard group configuration for each BEA MessageQ message queuing group. It provides memory-based storage of all undeliverable messages for the group that could not be stored for automatic recovery. The dead letter queue is defined as queue number 96 and named dead_letter_queue in the default group configuration information for each group. The default settings create this queue as a permanently active queue.

To use the dead letter queue, the sender program calls the `pams_put_msg` function specifying the appropriate `delivery` argument and using `PDEL_UMA_DLQ` as the `uma` argument. Any messages that cannot be delivered to the receiver program are written to the dead letter queue of the sender's group. An application program can attach to the queue named `PAMS_DEAD_LETTER_QUEUE` and use the `pams_get_msg` function to retrieve undelivered messages and use the `pams_put_msg` function to attempt redelivery.

An advantage of using the dead letter queue is the ability to recover undeliverable messages on a one-by-one basis. The sender program or another process within the application can attach to the DLQ and handle error recovery for each undeliverable message. A disadvantage of using the dead letter queue is the lack of disk storage for undelivered messages. A system failure on the sending node will cause all undelivered messages in the dead letter queue to be lost.

Using the Dead Letter Journal

When the `DLJ` UMA is used, the message is written to an auxiliary journal (the dead letter journal) if it cannot be delivered to the delivery interest point specified in the delivery mode argument. This UMA can only be used for recoverable messages. The `DLJ` UMA is used when the sender program needs to centralize error handling procedures and the application can support the resending of many messages from a file at a delayed interval. Storing undeliverable messages in a file ensures that they will not be lost if the system goes down, and allows redelivery attempts under user or application control.

A dead letter journal can be configured for each BEA MessageQ message queuing group. The dead letter journal provides disk storage for messages that could not be stored for automatic recovery. On BEA MessageQ for UNIX and Windows NT systems, a path name must be specified during configuration in order to create DLJ files. On BEA MessageQ for OpenVMS systems, DLJ files are created automatically by the MRS Server when a message queuing group is configured with MRS enabled.

To use the dead letter journal, the sender program uses the `pams_put_msg` function specifying the appropriate `delivery` argument and `PDEL_UMA_DLJ` as the `uma` argument. Any messages that cannot be stored by the message recovery system are written to the dead letter journal of the sender's group.

On UNIX and Windows NT systems, messages are recovered from the DLJ file using the dmqjplay utility. On OpenVMS systems, an application can provide recovery under program control or using system management tools. See your platform-specific installation and configuration guide for more information on how to use MRS utilities.

Using the SAF UMA

When the `SAF` UMA is used, the message is stored in the local journal file if the message recovery system is unable to store it in the remote journal file. The `SAF` UMA can be used with recoverable delivery interest points of DQF and CONF; however, it does not work with the `WF_SAF` delivery mode.

Use of the `SAF` UMA helps to manage the flow control between the sender and receiver systems. If the message cannot be written to the remote journal file due to insufficient resources or a cross-group link failure, the message will be written to the local journal file.

Note: The application must check the PSB UMA status value in order to know whether the message is recoverable.

Recoverable Messaging on BEA MessageQ Clients

Message Recovery Services (MRS) are also available for applications running on a BEA MessageQ client. The BEA MessageQ Client ensures delivery of recoverable messages to the Client Library Server on the BEA MessageQ Server by providing a store-and-forward (SAF) journal (`dmqsaf.jrn`) to store recoverable messages when the connection to a CLS is not available. Local SAF journal processing is available when Message Recovery Services (MRS) are enabled in the BEA MessageQ Client configuration. The location of the journal file is set when configuring MRS.

The Store And Forward journal (`dmqsaf.jrn`) is created on a BEA MessageQ client when MRS is enabled. The journal file is locked by the first application that attaches to the BEA MessageQ message bus. If you have several BEA MessageQ applications running on the client, only one can use the journal file. Other applications will get an error reading the journal when attaching and when sending to a queue. Each application program on the client requires a separate working directory. If there are many client applications running on a machine, consider configuring a message queuing group, which allows the applications to share resources.

If MRS is enabled, the message recovery journal is turned on when the client application first initiates an attach operation. If the CLS is not available at the time of an attach, the journal file is opened and the attach operation completes with return a status of `PAMS__JOURNAL_ON`.

When the journal is on, messages sent using the following reliable delivery modes are saved to the journal:

- `PDEL_MODE_WF_MEM` with `PDEL_UMA_SAF`

- `PDEL_MODE_WF_DQF`
- `PDEL_MODE_AK_DQF`
- `PDEL_MODE_WF_SAF`
- `PDEL_MODE_AK_SAF`

When the connection to the CLS is re-established, all messages in the SAF journal are sent before new messages are processed. The SAF messages are transmitted in first-in/first-out (FIFO) order. When the connection to CLS is reestablished, a return status of `PAMS__LINK_UP` is used to indicate that journal processing is no longer active.

Messages are sent from the SAF when one of the following events occurs:

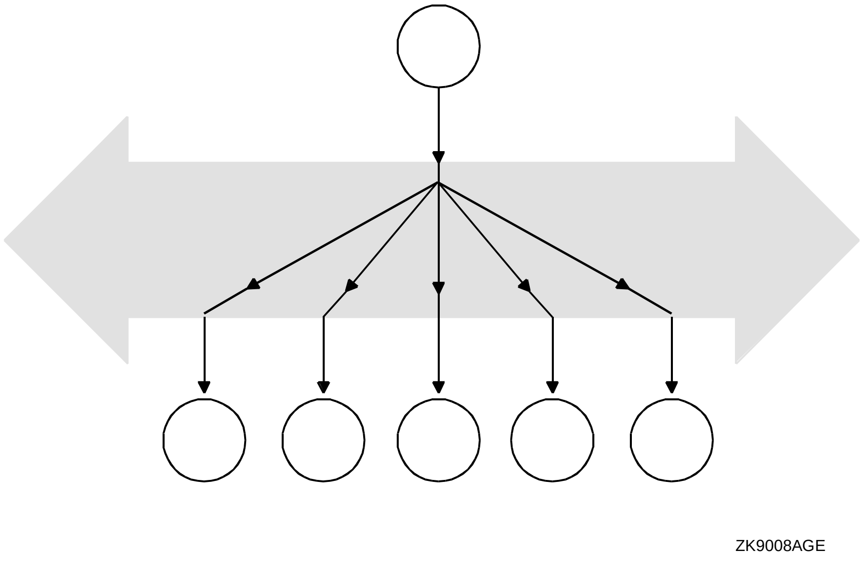
- The connection to the CLS is established successfully and pending messages exist in the SAF.
- The connection to the CLS is lost and the application continues to send recoverable messages. Additional message operations trigger an automatic reconnect to the CLS that is successful, and messages are pending transmission in the SAF.

CHAPTER

3 Broadcasting Messages

BEA MessageQ Selective Broadcast Services (SBS) enable applications to send a message to many receiver programs using a single program call. Any BEA MessageQ application can send a broadcast message using the standard `pams_put_msg` function. The sending application can generate broadcast messages without knowing the location or number of receiver programs.

Any BEA MessageQ application can selectively receive a broadcast message by first subscribing to a broadcast stream. To subscribe to a broadcast stream, the receiving application first sends a registration message to the SBS Server. Broadcast messages are then enabled for the application and flow into the receiver's queue for processing using the standard `pams_get_msg` function.



A broadcast stream is a data message pipeline that can have multiple entry points and multiple exit points. A message enters the stream and flows immediately to the end. There is no queuing on a broadcast stream, nor is the stream subject to flow control. Also, the flow of messages on a broadcast stream will not be interrupted by any event. Messages are only present on the stream for a finite segment of time, while they are being delivered to the queues of the receiving targets.

The SBS server is responsible for maintaining lists of user processes that are interested in broadcast streams. In addition, the SBS server is responsible for maintaining the various user definable rules that can be used to selectively extract messages from the broadcast stream that are set by the application using the `SBS_REGISTER_REQ` message.

Any BEA MessageQ application can send a broadcast message using the standard `pams_put_msg` function. The identical programming interface which is used to send point-to-point messages can also send a broadcast message by simply changing the target address to a Multipoint Outbound Target (MOT). A MOT is a broadcast stream associated with a queue number in the range of 4000 to 6000.

BEA MessageQ SBS works in a fashion similar to radio broadcasting. A BEA MessageQ sender program directs a message to a selected broadcast stream or “channel.” Then, the receiver program “tunes in” by sending a registration message to the SBS Server thus registering to receive messages broadcast over that channel. This feature is also called “publish and subscribe” in the messaging industry.

When a broadcast message is distributed, any receiver program registered for the broadcast channel will receive the message. Receiver programs that are not registered will not receive the message. Similar to radio broadcasting, where many radio stations are broadcasting at the same time, BEA MessageQ can distribute different types of messages over different broadcast channels.

SBS message broadcasting simplifies application development by eliminating the need for sender programs to know the number, state, or location of the target queues for each receiver program. SBS also simplifies application maintenance because receiver programs can be added and removed from the broadcast distribution without changing the sender program.

A common use for broadcast messaging is the display of real-time continuous data. For example, an application that provides up-to-date stock prices can obtain the latest values and display them simultaneously for any number of system users.

The following sections describe:

- How Message Broadcasting Works
- Sending Broadcast Messages
- Receiving Broadcast Messages
- Running Existing SBS Applications

How Message Broadcasting Works

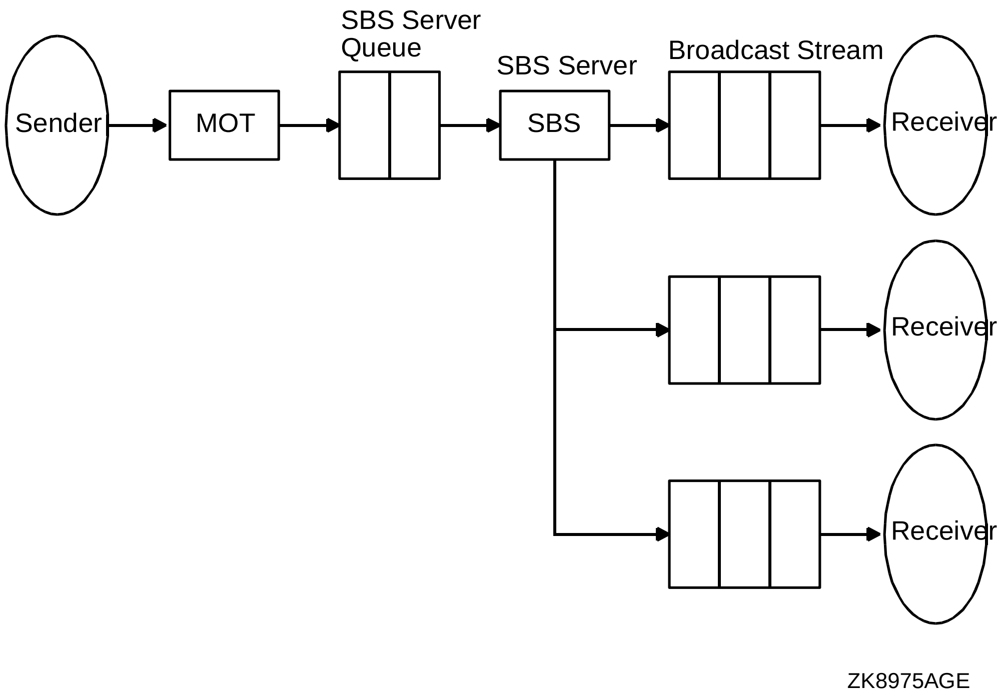
To send a message to multiple recipients simultaneously, the sender program uses the `pams_put_msg` function and specifies a Multipoint Outbound Target (MOT) as the target address for the message. A MOT, numbered between 4000 and 6000, is the identifier for a broadcast stream. A broadcast stream is the set of target queues registered to receive messages directed to a particular MOT.

Continuing our analogy with radio broadcasting, a MOT is equivalent to a radio station that people tune in to. When a message is sent to a MOT, any receiver program registered for the MOT will receive the message.

Each BEA MessageQ message queuing group can be configured to support message broadcasting by setting the `ENABLE_SBS` parameter in the Profile section of the group initialization file. The default value for this parameter is `YES`. Therefore, by default, an SBS Server is started for each message queuing group to support both message broadcasting and BEA MessageQ queue availability notification (AVAIL services). Receiver programs may register a queue address with any SBS Server. Any message directed to a MOT address is automatically redirected to the group's SBS Server.

The SBS Server uses its registration database to distribute the message to all applications that have registered to receive the selected message. Figure 3-1 illustrates the flow of messages in the broadcast stream.

Figure 3-1 BEA MessageQ Broadcast Stream



Once registered, applications can receive all messages directed to a broadcast stream, or only those messages that meet the selection criteria entered at the time of registration. Applications can register to receive messages from many broadcast streams. Applications can stop receiving broadcast messages at any time by sending an `SBS_DEREGISTER_REQ` message to the SBS Server.

For example, a stock brokerage application might need to display updated stock prices on many user terminals simultaneously. The system designer could designate MOT 5110 as the broadcast stream for updated stock prices. The sender program receiving the updated information from the stock exchange would create outbound messages containing the updated pricing information and send it to the broadcast stream represented by MOT 5110. During their initialization, all receiver programs designed to update user displays would send a registration message to their group's SBS Server requesting to receive all messages sent to MOT 5110. The updated stock price messages would then flow to the queue of the receiver programs.

When designing your broadcast communication environment, you can choose the following configuration characteristics:

- Private or universal broadcast streams
- Named or unnamed MOTs
- Message broadcasting using the standard BEA MessageQ transport (TCP/IP) or, on OpenVMS systems, the direct Ethernet multicast communication mode
- When using Ethernet broadcasting (OpenVMS only), you can choose between standard multicasting or the enhanced Recovery Protocol

The following sections describe:

- Broadcast Scope
- Named MOTs
- Broadcast Communication Modes

Broadcast Scope

The range of distribution for a broadcast stream is determined by the MOT address value. Table 3-1 lists the valid MOT address ranges:

Table 3-1 BEA MessageQ MOT Ranges

Type	Address Range	Description
Private MOT	4000-4999	Message distribution is restricted to the local group only.
Reserved	4900-5100	Reserved for use by BEA MessageQ. Of these addresses, the first 100 are local or private, and the second 100 are global or universal.
Universal MOT	5000-6000	Message distribution is to all SBS Servers

Any message sent to a queue address in the range of 4000-6000 is automatically redirected to the SBS Server. The broadcast queue address range (4000-6000) is divided into half, with the lower values designated as private MOT addresses and the higher as universal MOTs.

MOTs numbered below 5000 are associated with a **private broadcast stream**. MOTs numbered between 4900 and 5000 are reserved to for use by BEA MessageQ. BEA MessageQ redirects a message sent to a private broadcast stream to the local SBS Server, which restricts distribution to registered queues on that group. The SBS Server distributes the message by executing the rules for its local subscribers only. An application uses a private broadcast stream when the scope of interest for the information is local to one system.

An application program does not need to be local to an SBS Server group to register for a private broadcast stream. The registration message specifies a message group identifier allowing queues to register with remote SBS Servers.

MOTs numbered above 5000 are associated with **universal broadcast streams**. MOTs numbered between 5000 and 5100 are reserved for use by BEA MessageQ. In the universal MOT range, the broadcast stream is available to all SBS Servers. The sender SBS Server is responsible for the following:

- Ensuring that the submitted message conforms to the rule set when distributing the messages locally
- Distributing the messages to all partner SBS Servers

Each SBS Server in the BEA MessageQ network is responsible for ensuring that the submitted message conforms to the rule set of registered users and for distributing the messages locally.

For example, the stock brokerage application we mentioned may need to supply updated stock prices to receiver programs on many systems in a distributed network. This application would be most likely to use a universal broadcast stream to expedite the flow of information throughout the network.

Named MOTs

You can configure a BEA MessageQ MOT with a name so that the sender program can direct messages to the MOT name instead of the MOT number. To enable an application to refer to a MOT by name, define the MOT in the Group Name Table (GNT) section of the group initialization file. A full MOT address contains the following:

- The group ID in the high-order 16-bit word
- The MOT number in the low-order 16-bit word

The BEA MessageQ Naming Service supports the run-time lookup of MOT addresses by applications using the `pams_locate_q` function to translate a symbolic name to a MOT address. This name can have either a group-wide or bus-wide scope. The %GNT section of the group initialization file is used to load the name into the BEA MessageQ name space.

Listing 3-1 shows how to define the bus-wide "Alarm_events" name to use MOT 5110 and the group-wide "Operator_events" name to use MOT 4810.

Listing 3-1 Configuring a Named MOT

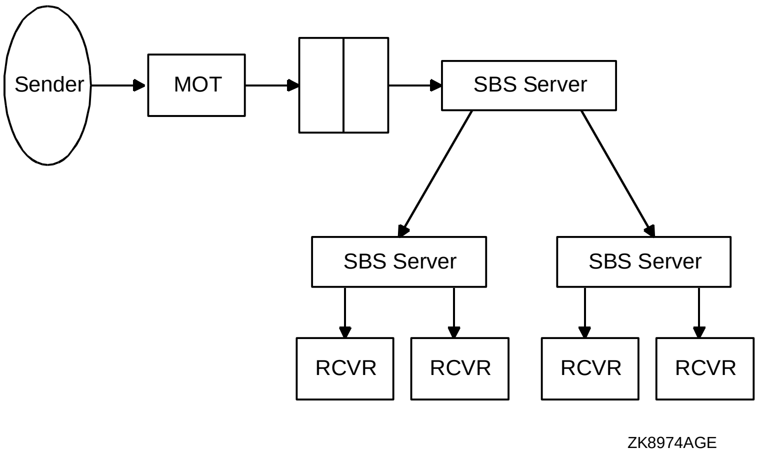
```
%GNT
!
! Name                 Group.Queue             Scope
-------------------    -----------             -----
Alarm_events               0.5110                G
Operator_events            0.4810                L
!
%EOS
```

Note that the group number is defined as 0 so that the application translating the name uses the local SBS Server rather than a specific SBS Server. See Chapter 4, “Using Naming” for a more detailed discussion on the use and features of BEA MessageQ Naming Services.

Broadcast Communication Modes

All BEA MessageQ Servers support message broadcasting using datagrams transmitted using the BEA MessageQ transport. Datagrams are then transferred over a BEA MessageQ cross-group link (TCP/IP) to another BEA MessageQ Server process and are queued to the receiving SBS Server. Since the broadcasting to each SBS Server is transmitted over point-to-point links, one copy of a message must be sent to each SBS Server. Datagram delivery mode can be used for both private and universal broadcast streams. Figure 3-2 illustrates SBS message broadcasting using the BEA MessageQ transport.

Figure 3-2 SBS Broadcasting Via BEA MessageQ Transport



On UNIX and Windows NT systems the default broadcast transport uses the standard BEA MessageQ cross-group messaging via TCP/IP as defined in the `%XGROUP` section of the group initialization file. On OpenVMS systems, the BEA MessageQ transport is specified in the SBS section of the `DMQ$INIT.TXT` file using the `COMM_SERVICE` keyword with `DG/DMQ` as the protocol and transport as shown in Listing 3-2:

Listing 3-2 Setting the COMM_SERVICE for SBS on OpenVMS

```
*             ---- Service ----
*               ID  Prot/Xport
COMM_SERVICE   10    DG/DMQ! default emulated broadcast path
```

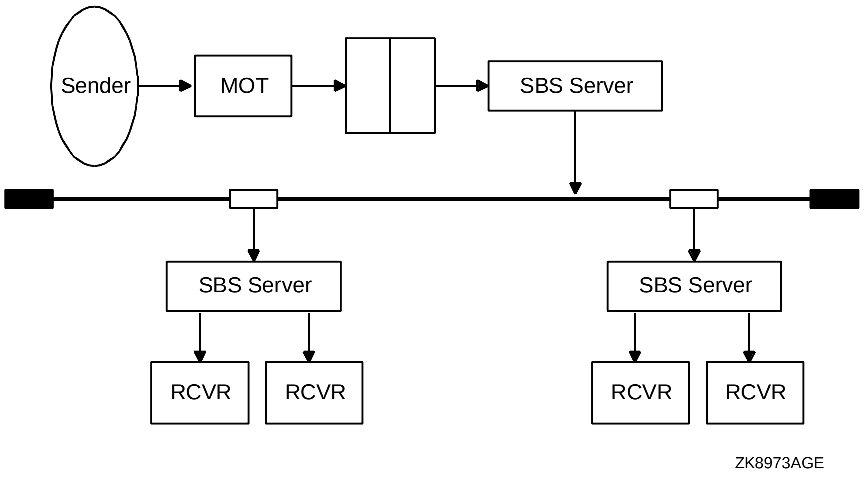
```
   GROUPS *           ! all known server groups
   REGISTER *         ! all universal MOTs
END_COMM_SERVICE
*
```

In addition to using the BEA MessageQ transport, BEA MessageQ for OpenVMS applications have the option to use Ethernet multicasting which provides faster throughput for message delivery. There are two protocols available for direct Ethernet multicasting, the normal datagram protocol or the enhanced recovery protocol. The default setting for broadcast communication between SBS Servers is provided by the BEA MessageQ transport using standard cross-group messaging. The choice of enhanced broadcast communication using Ethernet multicasting is set by the protocol parameter in the SBS Server Initialization section of the group initialization file.

Ethernet multicasting can only be used for universal MOT traffic. When Ethernet multicasting is enabled, a message to a universal MOT causes a datagram transfer to the SBS Server that transmits the message via an Ethernet multicast. All receiving SBS Servers obtain the multicast message directly. Since broadcasting utilizes the hardware multicast feature of the Ethernet device, a single multicast message can be received by any number of SBS Servers that are configured to listen using the multicast address as provided in the `CNTRL_CHAN` and `DATA_CHAN` keywords in the example below. The Ethernet DG protocol also supports simultaneous multicast on two Ethernet devices per system (also called dual-rail support). When dual-rail support is employed, message segments are broadcast on both Ethernet devices and duplicates are discarded by the receiving SBS Server.

If the optimized Ethernet multicasting feature is enabled, then MOT assignments to Ethernet physical addresses and protocol numbers must be specified. Figure 3-3 illustrates how SBS transports messages using Ethernet multicasting.

Figure 3-3 SBS Broadcasting Via Ethernet Transport



Retransmission Protocol on BEA MessageQ for OpenVMS Systems

BEA MessageQ for OpenVMS Version 4.0A provides an important enhancement to SBS Ethernet multicasting called the retransmission protocol. Each universal MOT that supports Ethernet multicasting can be configured with an option to retry transmission of cross-group messages in the event of delivery problems. Messages sent to a MOT that is configured for retransmission are stored in the SBS retransmission list after they are broadcast. The size of the retransmission list is configured in the %SBS section of the group initialization file. This size parameter sets the maximum number of messages stored by the SBS Server to fulfill retransmission requests in the event of message delivery failures.

The BEA MessageQ retransmission protocol divides Ethernet broadcast messages into the largest transportable segments possible and then transmits them to other SBS Servers. If a missing segment is detected, the receiving SBS Server requests retransmission of the message from the point at which the first missing segment was detected. This request is sent using a high-priority message to the sending SBS Server. The reply is returned using a high-priority message. If the message has already been deleted from the retransmission list, the sending SBS Server responds with a `NAK` message, generating a sequence gap notification for that MOT.

Ethernet multicasting and the retry option are enabled using the SBS Server Initialization section of the group initialization file. It contains a template for making these assignments when the group configuration is first customized. Listing 3-3 illustrates the configuration information that must be entered to SBS Server Initialization section of the group initialization file to configure Ethernet multicasting.

Listing 3-3 Configuring Ethernet

```
%SBS   ******* SBS Server Initialization Section ************
*
*   NOTE: Heartbeat interval is in units of 1 millisecond
*
HEARTBEAT       1000
*
*            ---- Service ----
*              ID  Prot/Xport
COMM_SERVICE   10    DG/DMQ! default emulated broadcast path
   GROUPS *! all known server groups
   REGISTER *! all universal MOTs
END_COMM_SERVICE
*
*            ---- Service ----
*              ID  Prot/Xport
COMM_SERVICE    0   DG/ETH ! datagram messaging over optimized Ethernet
   DEVICE_1  ESA0: ! VMS device name of the Ethernet board (rail A)
   DEVICE_2  EZA0: ! VMS device name of the Ethernet board (rail B)
   DRIVER_BUFFERS 16 ! # of VMS Ethernet driver buffers to preallocate[10-255]
   *
   *    <        <<<<<<<<<<<<<<<<< Warning >>>>>>>>>>>>>>>>>>
   * The protocol and Ethernet addresses show below are not registered
   * and are not guaranteed to be conflict free. Use them with discretion.
   *          |------ MCA ----|  |Prot #|
   CNTRL_CHAN AB-AA-34-56-78-90    81F0! used for VMS V2.x compatibility
   DATA_CHAN  AB-12-34-56-78-90    81F1! path for all data transmissions
   *
   *   NOTE: MAB = Message Assembly Buffer.  Each MAB requires area for
   *   a large message buffer, plus overhead of 150 bytes.
   *
   *                                                Default    Default   Heartbeat
   *              Transmit SILO  Receive SILO   Maximum     Poll     Dead Poll
   *         MOT    (in MABs)     (in MABs)    Heartbeat  Interval   Interval
   REGISTER 5101      30           15            4         10          10
   REGISTER 5102      35           12            4         10          10
   REGISTER 5156      10            6            6         10          10
   *
END_COMM_SERVICE
```

```
*
%EOS
```

When messages sent to a broadcast stream are distributed directly through Ethernet multicasting, it is important to monitor whether the application receives any sequence gap notifications. Because the queue storage area maintained by the hardware is small, messages can arrive faster than the I/O subsystem can deliver them. See the *BEA MessageQ Configuration Guide for OpenVMS* for a detailed description of how to configure Ethernet multicasting.

Sending Broadcast Messages

To broadcast a message, a sender program directs the message to the MOT that identifies the broadcast stream to use for message distribution. When the application issues the `pams_put_msg` function, BEA MessageQ recognizes the broadcast message because of the MOT address range and transparently redirects the message to the SBS Server of the target group for wider distribution.

Each message queuing group that is configured to distribute broadcast messages has an SBS Server associated with it. The SBS Server maintains a database of registered queues and message selection rules for each registered queue. The SBS Server compares each broadcast message against the rules stored for each registered queue and generates messages to all registered parties that meet the selection criteria.

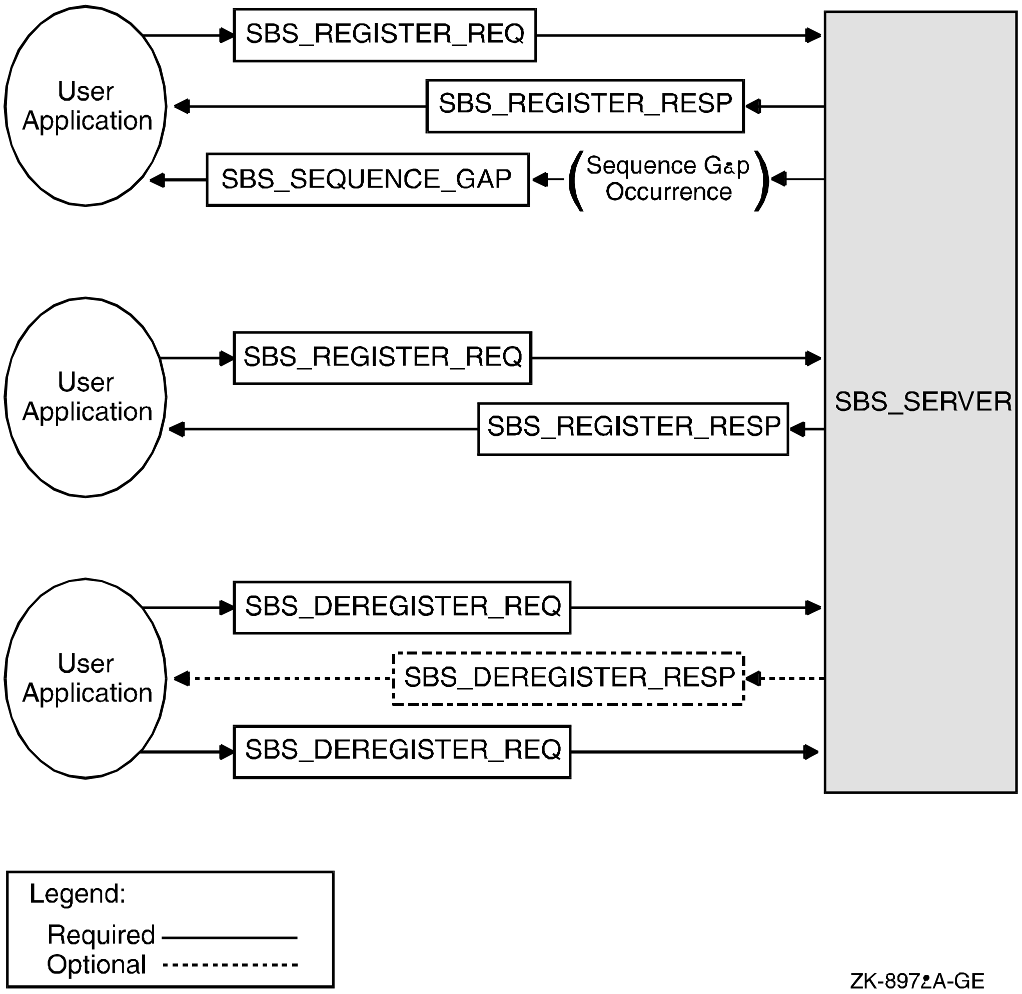
When a broadcast message is distributed by an SBS Server, the source field of the message is the MOT address identifying the broadcast stream. The target field is the registered target queue. The source address of the message's originator is obtained in the receiver program's show buffer argument to `pams_get_msg`. The SBS Server delivers only one copy of each message on the broadcast stream to each target queue, regardless of how many selection matches are made by separate subscription rule entries.

Broadcast messages cannot be stored for automatic recovery. However, you can configure the primary queue of the receiver program as permanently active to receive broadcast messages when the receiver program is not available. In addition, broadcast messages distributed using Ethernet multicasting now have limited recoverability through the retransmission protocol.

Receiving Broadcast Messages

To receive broadcast messages, applications use a standard set of BEA MessageQ messages to register for receipt with the SBS Server in their local group or in a remote message queuing group. Figure 3-4 illustrates the flow of messages sent to the SBS Server.

Figure 3-4 SBS Server Message Flow



The following sections describe:

- Registering to Receive Broadcast Messages
- Reading Broadcast Messages
- Deregistering from Receiving Broadcast Messages

Registering to Receive Broadcast Messages

To receive broadcast messages, an application registers a queue address with the SBS Server managing a broadcast stream. The queue address for the SBS Server in a message queuing group is queue number 99. Any BEA MessageQ primary, secondary, or multireader queue can be registered to receive broadcast messages.

Receiver programs register for broadcast messages using the `pams_put_msg` function sending a standard BEA MessageQ registration message. Typically, registration messages are sent to the primary queue of the local SBS Server queue (`SBS_SERVER`), which is queue 99 in the local group. The registration message contains the MOT of the broadcast stream plus any selection criteria related to messages that the application wishes to receive. An application can also register with a remote server by sending the registration message to the primary queue of the SBS Server in the remote group. (For example, 10.`SBS_SERVER`).

BEA MessageQ provides the `SBS_REGISTER_REQ` and `SBS_REGISTER_RESP` messages. Use `SBS_REGISTER_REQ` to request to register to receive broadcast messages. Your application receives the `SBS_REGISTER_RESP` in response to the `SBS_REGISTER_REQ` request message.

The registration information for each broadcast stream is stored in memory by each SBS Server and is volatile. Users registered with a remote SBS Server will no longer receive broadcast messages after the link to the remote server goes down. To recover from cross-group link failures, the application must monitor the status of the link to the remote SBS Server and be prepared to reregister for broadcast messages after a downed link is restored.

The receiver application can request sequence gap notification using the `SBS_REGISTER_REQ` message. The SBS Server maintains sequence checking on each broadcast stream. Sequence gaps occur when resource exhaustion and overflow conditions interrupt the reception of a broadcast stream by an SBS Server. For example, sequence gaps occur when a sender program broadcasts at a higher rate than the SBS Server can receive and distribute messages. When this characteristic is enabled, the SBS Server sends a message of type `SBS_SEQUENCE_GAP` to the target queue whenever a sequence gap is detected.

Sequence checking operates on the BEA MessageQ network and on the Ethernet LAN. On Ethernet, the channel and MOT number are returned in the sequence gap notification message. Broadcast messages are not recoverable; therefore, the occurrence of repeated sequence gap messages signals the need to synchronize application processing in the distributed network.

Broadcast streams hold messages for a short period of time only; therefore, receiver queues must be configured with a sufficient message receive quota to store messages as they arrive. As with any BEA MessageQ system, you must test the send and receive rates of programs to ensure that messages are not sent faster than they can be received.

The following sections describe:

- Sending a Registration Message
- Registering to Receive Selected Broadcast Messages
- Registration Acknowledgment

Sending a Registration Message

An application sends the registration message using the `pams_put_msg` function supplied with the following:

- The `target` argument as the queue address of the SBS Server from which the application wants to receive broadcast messages. The group number is the number of the remote group, or use zero to indicate the SBS Server in the local group. The SBS Server is defined as queue number 99 in the Queue Configuration Table of the default group initialization files.
- The `source` argument containing the queue number of the requesting application.
- The `class` argument as `MSG_CLAS_PAMS`.
- The `type` argument as `MSG_TYPE_SBS_REGISTER_REQ` to receive all messages from a broadcast stream.

The message data structure of the registration message contains the address of the broadcast stream from which the application wants to receive messages and the address of the target queue address to receive broadcast messages.

Registering to Receive Selected Broadcast Messages

Use the `SBS_REGISTER_REQ` message to register for selective reception of broadcast messages. This message registers a target queue to receive a copy of all messages on a broadcast stream that meet a single selection rule.

The selection rule requests the SBS Server to compare an operand in the message header or message data structure with the operand supplied in the selection rule. The term operand refers to the data in the message header or message data structure that will be compared. For example, a selection rule may configured to receive only messages with a particular type code. In this case, the message type code is the operand. The `SBS_REGISTER_REQ` message can define up to 255 selection rules. Message distribution can be made if **any** or **all** of the selection rules are found to be true.

A selection rule is composed of the following components:

- Data Offset
- Operator
- Operand Length
- Operand Field

Data Offset

The data offset field indicates whether the selection criteria is part of the message header or the message area. If the data offset is a positive value or zero, then this message is used to begin the comparison. BEA MessageQ specifies constants for selection based on the type, class, or sending queue. Matching based on message priority is not supported. Table 3-2 lists the data offset symbols.

Table 3-2 Valid Data Offset Symbols

Offset	Description
`PSEL_CLASS`	Use the class field of the message
`PSEL_TYPE`	Use the type field of the message

Table 3-2 Valid Data Offset Symbols

Offset	Description
`PSEL_SOURCE`	Use the source field of the mesage. **Note:** The comparison is made to the original source field, not the MOT address.
`MATCH_PRIORITY`	Not supported
zero based data offset	Use the message offset to begin data comparison

Operator

The operator field indicates the type of comparison to be performed on the operands. Table 3-3 lists the symbols for the operator field.

Table 3-3 Operator Field Symbols

Operator	Description
`PSEL_OPER_ANY`	Always match
`PSEL_OPER_EQ`	Equal
`PSEL_OPER_NEQ`	Not equal
`PSEL_OPER_GTR`	Greater than
`PSEL_OPER_LT`	Less than
`PSEL_OPER_GTRE`	Greater than or equal to
`PSEL_OPER_LTE`	Less than or equal to
`PSEL_OPER_AND`	Operand field AND data not equal to zero

Operand Length

The operand length field specifies the number of bytes in the operand field to be used for comparison. The operand length can be 1, 2, or 4 bytes only.

Operand Field

The operand field is the value to be compared with the selected field in the message header or message data structure.

Registration Acknowledgment

The `SBS_REGISTER_RESP` message replies to the `SBS_REGISTER_REQ` request. This response message returns a status indicator, the registration ID, and the number of application queues registered to receive messages from the broadcast stream.

Reading Broadcast Messages

When a message is sent to a broadcast stream, the SBS Server uses its registration database to determine which applications have registered to receive that kind of message. The SBS Server automatically sends the messages to the distribution of all matching applications. The receiving application reads the broadcast message from its target queue using the pams_get_msg, pams_get_msgw, or pams_get_msga functions. The source of the message, as seen by the receiving application, is the broadcast stream. The address of the sender is also provided to the receiving application in the 'original source' field of the PAMS show buffer.

Deregistering from Receiving Broadcast Messages

An application can withdraw from the broadcast stream by either sending the `SBS_DEREGISTER_REQ` deregistration message to the SBS Server, or by exiting the BEA MessageQ message queuing bus when the automatic deregistration was previously set in the subscription entry. Either of these actions removes the subscription entry from the internal SBS tables. Temporary queues are automatically deregistered when the application exits.

Applications that use the deregistration message can request subscription cancellation in one of the following ways:

- Cancel by exact match of the MOT address and target queue
- Cancel by subscription ID

Sending a message of type SBS_DEREGISTER_REQ causes the SBS Server to deregister all entries for the broadcast stream and target queue. If requested, an SBS_DEREGISTER_RESP message will acknowledge the SBS Server deregistration.

To cancel registration for a specific type of message while continuing to receive other broadcast messages, the application must send a message of type SBS_DEREGISTER_REQ using the subscription identification code assigned to the original SBS Server registration. Deregister by ID if there is more than one registration for the broadcast stream and target queue and you only want one entry to be removed.

Running Existing SBS Applications

Applications using SBS messages that were designed to run under BEA MessageQ for OpenVMS, Version 3.2 or earlier will continue to run under BEA MessageQ for OpenVMS, Version 4.0A. For BEA MessageQ V4.0, the SBS message interface was redesigned to support enhanced features and to make the message structures RISC aligned.

However, to run these applications in other BEA MessageQ environments such as UNIX or Windows NT, the applications must be changed to use the new Version 4.0 SBS messages. Table 3-4 is a list of the new SBS messages and their obsolete equivalent message. See the detailed description of each message in Table 3-4 to learn the changes needed to recode your application to use the new SBS messages.

Table 3-4 Obsolete and New SBS Messages

Obsolete SBS Messages:	New SBS Messages:
SBS_BS_SEQGAP	SBS_SEQUENCE_GAP
SBS_DEREG	SBS_DEREGISTER_REQ
SBS_DEREG_ACK	SBS_DEREGISTER_RESP
SBS_DEREG_BY_ID	SBS_DEREGISTER_REQ
SBS_REG	SBS_REGISTER_REQ
SBS_REG_EZ	SBS_REGISTER_REQ

Table 3-4 Obsolete and New SBS Messages

Obsolete SBS Messages:	New SBS Messages:
`SBS_REG_REPLY`	`SBS_REGISTER_RESP`
`SBS_REG_EZ_REPLY`	`SBS_REGISTER_RESP`

CHAPTER

4 Using Naming

Naming is a powerful feature that enables BEA MessageQ applications to identify message queues by name whether they reside on the local system or on another system on the BEA MessageQ message queuing bus. Naming also allows applications to bind permanent and temporary queues to names at runtime.

Application developers use the BEA MessageQ naming feature to separate their applications from the underlying BEA MessageQ environment configuration. By referring to message queues by name in their applications, developers do not have to modify their applications when the BEA MessageQ environment configuration changes. A name can also be associated with a Multipoint Outbound Target (MOT) address when broadcasting messages.

The following sections describe:

- Understanding Naming
- How to Configure Bus-wide Naming
- How Applications Use Naming
- Static and Dynamic Binding of Queue Addresses

Understanding Naming

Before you can use naming, you need to understand the following key concepts in using BEA MessageQ naming:

- What is Naming?
- What is a Name space?

- What is the Naming Agent?

What is Naming?

The BEA MessageQ naming feature enables applications to refer to message queues by name. These names are also called **queue references**. The queue reference and its associated queue address must be defined to BEA MessageQ, either statically in the group configuration file, or dynamically using the `pams_bind_q` function. The `pams_locate_q` function performs the name-to-queue address translation at runtime.

When a name or queue reference is defined it is assigned a **scope**. Names can be assigned a “group-wide” scope to enable the name to be used by any application running in that message queuing group (local queue reference). Names can be assigned “bus-wide” scope to enable any application on the message queuing bus to refer to the queue by name (global queue reference).

What is a Name Space?

A **name space** is the repository where names and their associated queue addresses are stored. When an application refers to a queue by name, BEA MessageQ must look up the name in the name space to find its associated queue address in order to send a message to the named queue.

BEA MessageQ uses three levels of name spaces: process, group, and bus. Names are stored in the group- or bus-wide name space whether their configuration scope defines a local or global queue reference. The process name space is an application cache used to improve performance. Names can exist in one or all three of the name spaces. However, they are defined only in one of these spaces and can be cached at different levels. Users can bypass caching when they use `pams_locate_q` if they favor accuracy over performance.

When a group starts up, it creates the group-wide name space and populates it with entries defined in the `%QCT` and `%GNT` sections of the group initialization file. In addition, entries configured in the `%QCT` and `%GNT` sections with a global scope are updated in the bus-wide name space. In order to use bus-wide naming, you must configure your environment to use this BEA MessageQ feature.

BEA MessageQ offers two types of name spaces:

- *Light weight* — this type of name space is included with BEA MessageQ. The BEA MessageQ lightweight name space uses a directory structure shared among naming agents. When two nodes view the name space, it must be exactly the same. In this way, BEA MessageQ can deal with any coordination automatically. Some examples of shared file systems are clusters and NFS-mounted disks.
- *Heavy weight* — this type of name space is offered by an add-on product to BEA MessageQ which has its own server and spans the entire network. Currently, the only heavy weight name space supported by BEA MessageQ is DECDNS. Although naming agents servicing DECDNS can only run on the OpenVMS platform, DECDNS names can be bound or located by applications running on any BEA MessageQ supported platform, including client implementations.

What is the Naming Agent?

The Naming Agent is the BEA MessageQ process that accesses and manages the BEA MessageQ bus-wide name space. Users configure groups to decide whether a group hosts or remotely accesses a Naming Agent. When a group starts, it launches the Naming Agent, if it is hosted by this group.

When a group starts up, the BEA MessageQ startup procedure requests the Naming Agent to update all entries in the initialization file that have a global scope.

Applications do not access the bus-wide name space directly; when an application uses a global queue reference, it is the Naming Agent that looks up the name in the bus-wide name space and returns the queue address to the application.

Configuring Bus-Wide Naming

The use of group-wide naming requires no special configuration steps because the process-level name space is created by attaching to the BEA MessageQ message queuing bus and the group-wide name space is created by the group control process. To use local (group-wide) naming, configure queue names in the Queue Configuration Table (`%QCT`) or the Group Name Table (`%GNT`) section of the group initialization file.

When the group starts up, BEA MessageQ automatically creates the group name space. It creates the process name space when an application attaches to the message queuing bus.

To enable your applications to use global (bus-wide) naming, you must perform additional configuration steps. First, you must decide the group or groups in which the naming agent will run. BEA MessageQ allows you to specify a main group and an alternate group to run the Naming Agent. The BEA MessageQ Naming Agent is the BEA MessageQ Server that maintains the namespace for name-to-queue address translations and performs the runtime queue lookup when an application refers to a queue by name.

The `%NAM` section of the group initialization file enables you define the group or groups in which the Naming Agent process will run. BEA MessageQ allows the definition of two naming agents for each message queuing bus. When BEA MessageQ starts each group, it looks in this section of the initialization file to decide whether to start a naming agent for the group. For groups that do not run a Naming Agent, BEA MessageQ uses the information in the `%NAM` section to direct requests to the Naming Agent. Groups must have a cross-group connection to the groups in which the Naming Agent runs.

To use global naming, you must create a namespace on the nodes on which the Naming Agents will run. BEA MessageQ enables users to configure two Naming Agents to support global messaging for the environment. In order to allow the second Naming Agent to form a backup for the first, both Naming Agents must be configured to use the same name space. Therefore, when you configure your name space for use by two Naming Agents that run on different systems, it must use a shared file system that is accessible to both Naming Agents.

To use a global name, at least some portion of the path name must be specified. Path information can be supplied by the application, or you can use the `DEFAULT_NAMESPACE_PATH` parameter in the `%PROFILE` section of the group initialization file in order to create and maintain path information for global names. For global naming to function properly, this parameter must be set to the same value for all groups in which applications are designed to access the same name space. When the naming agent is enabled in the group initialization file, a file *uid*.dnf is created in the `DEFAULT_NAMESPACE_PATH` directory which contains the global names. The following syntax shows how to set the default namespace to be created and maintained in the name space.

Use the Queue Configuration Table (`%QCT`) or the Group Name Table (`%GNT`) of the group initialization file to create static or dynamic definitions for global names as follows:

- Define global static names in the `%QCT` or `%GNT` by providing the name, the queue address and setting the name scope identifier to `G` for global names.
- Define global dynamic names by supplying the name, `0.0` as the address and the `G` identifier for global names. Names defined with a `0.0` address can be dynamically bound to a queue address at runtime using the `pams_bind_q` function.

For a detailed description of how to configure your environment and develop applications to use global naming, refer to the installation and configuration guide for your platform.

How Applications Use Naming

Queues and local queue references exist in groups, which exist in buses. Global queue references can exist anywhere in the bus-wide name space. Applications in all groups can bind and look up global queue references.

The set of directory names from the root of the hierarchy to where the queue is defined is called its path. The path plus the queue's name is called its pathname. A name must be unique within its directory. Thus, any queue can be uniquely identified by its pathname.

Queues and local queue references are always identified by their names. A global queue reference must be identified by its pathname. However, it can be identified by its name only if its path is the group's `DEFAULT_NAMESPACE_PATH`. (The `DEFAULT_NAMESPACE_PATH` is set in the `%PROFILE` section of the group initialization file.)

The following sections describe:

- Specifying Names and Pathnames
- Attaching and Locating Queues

Specifying Names and Pathnames

BEA MessageQ applications can be developed to be independent of the bus-wide name space implementation for a particular environment. This means that no coding changes are required if the application environment initially uses the BEA MessageQ lightweight name space and migrates to a heavy weight name space at a later time.

Names are specified in BEA MessageQ applications in one of three ways:

- fully qualified
- partially qualified
- unqualified

For detailed information on how to specify path names and file names, refer to the installation and configuration guide for the platform that you are using.

Attaching and Locating Queues

An application may only read messages from queues in its own group. To read from a queue, an application must attach to the queue using the `pams_attach_q` function. For a permanent queue, it must identify the queue by its name, its address, or a queue reference. For a temporary queue, the attach operation creates the queue and assigns it an address.

An application can send messages to a queue in its own group and to queues in other groups. When sending a message, the target queue is always identified by its address. An application can directly code in the address, or it can use the `pams_locate_q` function to derive the queue’s address from its name or queue alias. When `pams_locate_q` is used with `pams_put_msg`, applications can remain separate from the details of system configuration because they are able to obtain the physical address of the target queue at runtime.

Static and Dynamic Binding of Queue Addresses

BEA MessageQ offers two approaches to associating a queue reference (also called a queue name) with a queue address: static and dynamic. **Static binding** refers to associating a queue name with a queue reference using the queue configuration table (`%QCT` section) and the group name table (`%GNT` section) in the group initialization file. Static binding creates the association when the group starts up.

Dynamic binding refers to the use of the `pams_bind_q` to associate a queue name with a queue address after the application starts up. With dynamic binding, you can write applications that dynamically "sign up" to service a queue at runtime. This means that your application can access a service without having to be aware that its normal host computer is down and that the service is being provided from another host computer. An application does this by dynamically associating a queue address to a queue reference at run-time.

The following sections describe:

- How Dynamic Binding of Queue Addresses Works
- How Caching and Binding Works
- Examples of Static and Dynamic Binding

How Dynamic Binding of Queue Addresses Works

Dynamic binding of queue addresses allows you to share queue names with any application attached to the message queuing bus. An application can attach to a queue in a group and bind this queue to a name into the bus-wide name space so that an application in another group can locate this queue in the bus-wide name space and send messages to it.

BEA MessageQ provides the `pams_bind_q` function, which associates a queue address to a queue name at runtime. The `name_space_list` argument in the `pams_attach_q` and `pams_bind_q` functions identify the scope of the queue name and controls cache access.

How Caching and Binding Work

When an application process locates a name for the first time, it is cached in the process name space. If the name is for a global queue reference, it is also cached in the group name space. Conversely, later lookups can fetch the name from the "nearest" location that holds the name. For example, suppose `APP1` in a group locates **`gqref1`**. This causes **`gqref1`** to be cached in `APP1`'s process name space and in the group. Also, suppose `APP2` in this group locates **`gqref1`**. Since `APP1`'s process cache is invisible to `APP2`, `APP2` will fetch **`gqref1`** from the group.

When an application process deletes a queue that is bound to a reference or when it binds a reference to a new address, BEA MessageQ automatically updates the applications process cache, the name's entry in the group, and (if this is a global queue reference), the global name space also. However, other places where this queue is cached are not updated.

When your application detaches or exits from a queue that was bound to a name, BEA MessageQ unbinds the queue before exiting or detaching.

Examples of Static and Dynamic Binding

You can code your application to use either static or dynamic binding of queue addresses. Use static binding if the queue that your application attaches to is not going to change its address (for example, a permanent queue). Otherwise, if the queue that your application needs may change (for example, if the queue is temporary, or if the application runs in different groups), code your application to use dynamic binding of the queue address.

When coding, keep in mind that there are two name-based queue identification styles that you can use. They are as follows:

Client for Style 1 (Static Binding)

Listing 4-1 is a pseudocode fragment showing static binding of a queue address for a client.

Listing 4-1 Client Style Static Binding

```
    pams_locate_q("gqref1", q_address,
       [PSEL_TBL_PROC, PSEL_TBL_GRP, PSEL_TBL_BUS] )
 loop:
    build request message
 putloop:
    status = pams_put_msg(q_address)
    if status is error,
        print descriptive error
        pause and goto putloop,
        or exit program as desired
    goto loop
```

Client for Style 2 (Dynamic Binding)

Listing 4-2 is a pseudocode fragment showing dynamic binding of a queue address for a client. In this example, when an error occurs, the client attempts to see if a new server has signed up to provide this service. Note that it does not use the cache when it refinds `gqref1` because it wants to see the binding established by the new server, not the out-of-date cached binding.

Listing 4-2 Client Style Dynamic Binding

```
     pams_locate_q("gqref1", q_address,
             [PSEL_TBL_PROC, PSEL_TBL_GRP, PSEL_TBL_BUS]
)
 loop:
    build request message
 putloop:
    status = pams_put_msg(q_address)
    if status is error then
       pams_locate_q("gqref1", q_address1, [PSEL_TBL_BUS])
    if q_address not q_address1 then
       q_address = q_address1
       goto putloop
    else pause and goto putloop, or exit program as desired
```

Server for Style 1 (Static Binding)

Listing 4-3 is a pseudocode fragment showing static binding of a queue address for a server.

Listing 4-3 Server Style Static Binding

```
    pams_attach_q("gqref1", q_address, PSYM_ATTACH_BY_NAME,
               [PSEL_TBL_PROC, PSEL_TBL_GRP, PSEL_TBL_BUS])
 loop:
    pams_get_msg(q_address)
    process request and reply
    goto loop
```

Server for Style 2 (Dynamic Binding)

Listing 4-4 is a pseudocode fragment showing dynamic binding of a queue address for a server. In this example, the server attaches to a queue and then tries to make this queue the provider of the `gqref1` service. However, if another server is already providing the service, the program exits.

Listing 4-4   Server Style Dynamic Binding

```
  pams_attach_q(any attach options, q_address)
  status = pams_bind_q("gqref1", q_address, [PSEL_TBL_BUS])
  if status = "queue reference already bound to a queue"
  then exit program
loop:
  pams_get_msg(q_address)
  process request and reply
  goto loop
```

CHAPTER

5 Using Message-Based Services

BEA MessageQ applications regularly perform standard tasks such as checking the state of a queue or the status of a cross-group connection before sending a message. To make these tasks easier, BEA MessageQ offers message-based services, which are sets of predefined request, notification, and response messages exchanged between the application and BEA MessageQ server processes.

Table 5-1 describes the functions performed by using message-based services and lists the servers they are available through.

Table 5-1 Overview of Message-Based Services

You can . . .	Through the . . .
Obtain the status of a particular queue	Avail Server
Monitor and control link status	Connect Server
Obtain the current status of all queues	Queue Server
Register for broadcast messages	SBS Server
Manage message recovery files (OpenVMS systems only)	MRS Server
Transfer messages from one DQF file to another (OpenVMS systems only)	Qtransfer Server

How Message-Based Services Work

BEA MessageQ uses message-based services to perform routine tasks such as obtaining queue status. There are two request-response paradigms used by message-based services. For some kinds of services, the sender program sends a request to a BEA MessageQ server using a particular message. The BEA MessageQ server returns the response in a message using a particular message type and format. If information was requested, it is returned in the message area of the response message.

In other cases, a sender program may register to receive ongoing updates of information. In this case, the sender program sends a registration request and receives a response if the registration request is successful. In addition, the sender program receives event-driven messages providing up-to-date information as requested. To stop receiving the event-driven messages, the sender program must send a deregistration request to the BEA MessageQ server.

Service requests are directed to the primary queue of the BEA MessageQ server designated to provide the selected service. BEA MessageQ message-based service requests are delivered to BEA MessageQ servers using the BEA MessageQ application programming interface (API) or BEA MessageQ scripts. Similarly, applications obtain response and notification messages by reading these messages from their primary or response queue.

BEA MessageQ message-based services are sent between a user application program that functions as a requestor and a BEA MessageQ server process that fulfills the request. For messages to be properly understood between systems, message data must be sent and returned in the endian format understood by both the requestor and the server.

Most BEA MessageQ message-based services automatically perform this conversion if the endian format of the two systems is different. However, some message-based services do not perform this conversion. Therefore, the user application must convert the message to the endian format of the server system to ensure that the message data is correctly interpreted.

See the description of each message for information on whether BEA MessageQ performs the conversion or the application must check for differences in hardware data formats. See the Building and Testing Applications topic to learn how you can ensure that your application formats data properly and performs required conversions when sending standard messages between computer systems from different vendors.

Requesting a Service

You can send a service request message using the pams_put_msg function. Request messages use the **type** argument to identify the purpose of the message. Each request message has a predefined data structure.

To send a standard request message, supply the following:

Target	The symbolic name for the BEA MessageQ server fulfilling the request. For example, use `PAMS_AVAIL_SERVER` for requests handled by the BEA MessageQ Avail Server process.
Class	The class code PAMS indicating that the message is a BEA MessageQ message-based service request.
Type	The type code of the message you are sending. For example, `AVAIL_REG`.
Message data	The predefined data structure containing the information to be sent with the service request. The definition of all BEA MessageQ message-based services messages is now provided in the `p_msg.h` include file.

A detailed description of each message in the Message Reference topic explains each field in the data structure and provides a sample C message structure.

Receiving a Response

Each BEA MessageQ server returns response or notification messages to answer a service request. Most request messages have a response message. In addition, some service requests are answered by the BEA MessageQ server with a notification message that supplies information to the sender program as it becomes available.

When an application requests information using the `pams_put_msg` function, it provides the BEA MessageQ server with the group ID and queue number to which the response should be directed. The sender program then reads this queue using the `pams_get_msg`, `pams_get_msgw`, or `pams_get_msga` function to obtain the response information.

A BEA MessageQ server response and notification message provides the following:

Source	The symbolic name of the BEA MessageQ server fulfilling the request.
Class	The class code of the response is always PAMS, indicating that this is a BEA MessageQ message-based service.
Type	The type code of the message received. For example, `AVAIL_REG_REPLY`.
Message data	The predefined data structure used to provide requested information in the response or notification message. The definition of all BEA MessageQ message-based services messages is now provided in the `p_msg.h` include file.

A detailed description of each message in the Message Reference topic explains each field in the data structure and provides a sample C message structure.

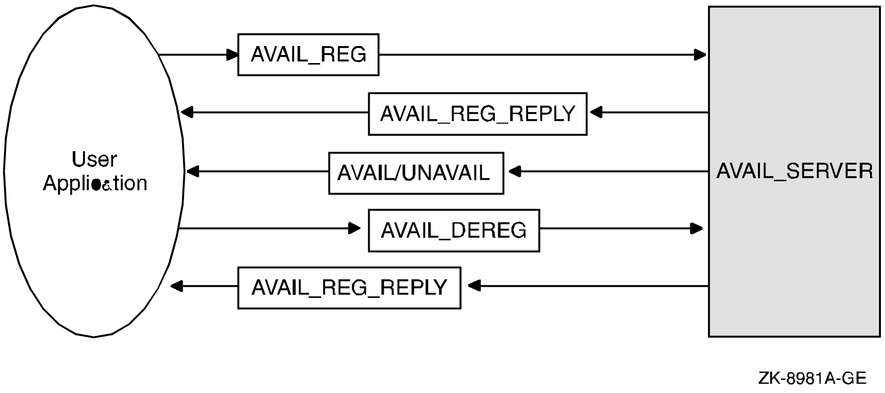
Obtaining the Status of a Queue

BEA MessageQ message-based services enable applications to check whether a particular queue is available to receive messages. This set of messages returns information on the status of any active queue in a local or remote group.

To obtain information on the status of a particular queue, applications exchange the following messages with the Avail Server:

- `AVAIL_REG`—Request message to register to receive queue information.
- `AVAIL_REG_REPLY`—Response message to confirm registration or deregistration.
- `AVAIL`—Notification message to indicate that the queue is available.
- `UNAVAIL`—Notification message to indicate that the queue is unavailable.
- `AVAIL_DEREG`—Notification message to deregister from obtaining queue information.

Figure 5-1 Avail Server Message Flow



An application program registers to receive availability messages by sending a message of type `AVAIL_REG` to the local Avail Server process. The Avail Server responds with a message of type `AVAIL_REG_REPLY`, acknowledging the notification request.

After registration, the requestor immediately receives an `AVAIL` or `UNAVAIL` message indicating the current availability of the target queue. Queue availability messages provide ongoing notification when a specific queue becomes attached or detached and when a link is connected or lost. If the queue becomes active because a process becomes attached, the Avail Server sends a message of type `AVAIL`. If it becomes inactive, the server sends a message of type `UNAVAIL`.

Applications must cancel availability notification by sending a message of type `AVAIL_DEREG`. The application receives a `AVAIL_REG_REPLY` message indicating the status of the operation. It is important to note that if the distribution queue for an AVAIL registration becomes unavailable, the registration will be automatically deleted by BEA MessageQ. A subsequent attempt to deregister AVAIL services for this distribution queue will result in an error message indicating that the registration does not exist.

Monitoring and Controlling Link Status

This section describes how applications can use BEA MessageQ message-based services with the Connect Server process to obtain information on connections, queue entries, groups, cross-group connections, and link status.

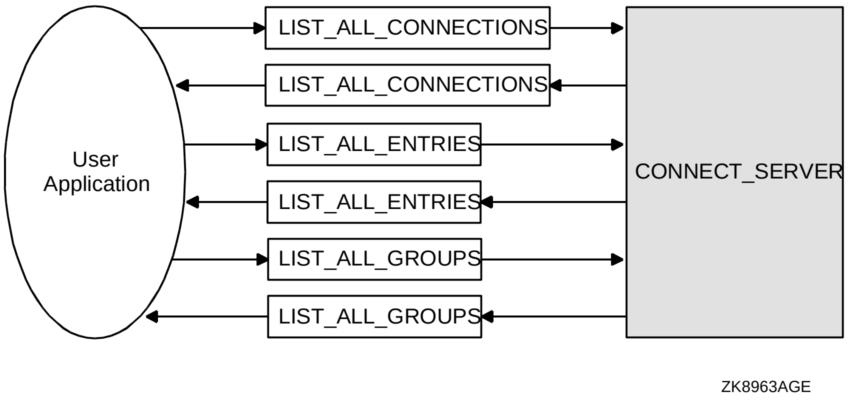
Listing Cross-Group Connections, Entries, and Groups

An application can request a list of current cross-group connections or all configured cross-group entries from the Connect Server. This request allows the application to obtain the current BEA MessageQ cross-group configuration and active cross-group connections. In addition, the Connect Server can provide a list of known queues in a group and a list of all groups defined on a message queuing bus.

To obtain a list of all cross-group connections, configured groups, and queue entries, applications exchange the following messages with the Connect Server:

- `LIST_ALL_CONNECTIONS` (Request)—Request message to provide a list of all cross-group connections.
- `LIST_ALL_CONNECTIONS` (Response)—Response message to provide a list of all cross-group connections. Groups with no link connection are not listed.
- `LIST_ALL_ENTRIES` (Request)—Request message to provide a list of all queue entries for a group.
- `LIST_ALL_ENTRIES` (Response)—Response message to provide a list of all queue entries for a group.
- `LIST_ALL_GROUPS` (Request)—Request message to provide a list of groups on the message queuing bus.
- `LIST_ALL_GROUPS` (Response)—Response message to provide a list of all groups, connected and unconnected, on the message queuing bus.

Figure 5-2 Requesting Cross-Group Information



To obtain a list of all groups defined on the message queuing bus, send a `LIST_ALL_GROUPS` message to the Connect Server. To obtain a list of all cross-group connections for the message bus or a list of all cross-group entries, send a `LIST_ALL_CONNECTIONS` message to the Connect Server. To obtain a list of queues in a group, send a `LIST_ALL_ENTRIES` message.

The reply to these requests is a variable-length message with the same type and class as the request. To read the information returned, the application uses the message size parameter returned by the pams_get_msg function and divides it by the byte size of the data object requested to determine the number of data entries returned. The byte size of these entries is described in the reference description of each message.

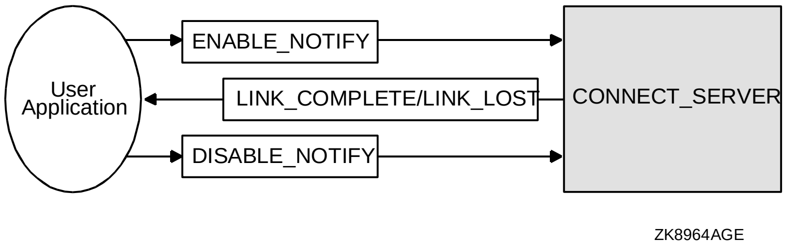
Obtain Notification of Cross-Group Links Established and Lost

An application can also use Connect Server messages to receive notification of cross-group links connected and disconnected in its own group. To obtain information on the status of cross-group links, use the following message-based services:

- `ENABLE_NOTIFY`—Request message to request notification of link changes.
- `LINK_COMPLETE`—Notification message to indicate that the cross-group link was created.

- `LINK_LOST`—Notification message to indicate that the cross-group link was lost.
- `DISABLE_NOTIFY`—Request message to request disabling of link change notification.

Figure 5-3 Requesting Cross-Group Link Status



Applications send an `ENABLE_NOTIFY` message to the Connect Server to receive ongoing notification when new connections are made or lost. Registered applications receive a `LINK_COMPLETE` notification message when a new cross-group connection is created. Applications receive a `LINK_LOST` message when a cross-group connection is lost. To deregister from receiving further notification messages, the application sends a `DISABLE_NOTIFY` message to the Connect Server.

Note: To receive ongoing notification of queue attachments, we recommend the use of the Queue Server messages, such as `ENABLE_Q_NOTIFY_REQ`. The `ENABLE_NOTIFY` message should no longer be used to obtain queue attachment information.

Controlling Cross-Group Links

In addition to obtaining information on cross-group links, the Connect Server messages can be used to control cross-group connections through a feature called link management. Applications use link management messages to explicitly control the creation and deletion of cross-group links. Explicit control over remote links may be required by an application to restrict network communication with a particular node or to reduce network traffic.

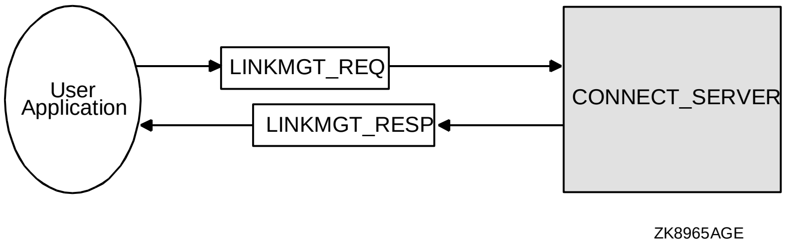
The `LINKMGT_REQ` request message enables the following control functions:

- Inquire—Allows querying of a group's link state.

- Enable—Re-enables a link's address entries.
- Disable—Disables a link's address entries.
- Connect—Re-enables a link's address entries and connects to selected groups.
- Disconnect—Implicitly disables links and disconnects links to requested groups.

The `LINKMGT_RESP` response message notifies the requesting application if the request was successful and supplies information about the cross-group connection. Link management functions are also available through the System Manager utility on BEA MessageQ for OpenVMS systems. Figure 5-4 is a graphical representation of the functional relationship facilitated by `LINKMGT_REQ` and `LINKMGT_RESP`:

Figure 5-4 Using Link Management



Link management can also be event driven. For example, an application event can trigger a link to another group, which enables message exchange.

Note: When using link management, automatic creation of cross-group connections must be disabled with the generate connect option `D` (disable) in the `%XGROUP` section of the BEA MessageQ group initialization file to completely control all cross-group links. For more information, refer to the Enabling Network Connections in the Cross-Group Section topic in the BEA MessageQ Installation and Configuration Guide for each platform.

Link Management Control Functions

The link management request message allows for the following control functions:

- Inquire—Allows querying of a group's link state.
- Enable—Re-enables a link's address entries.

- Disable—Disables a link's address entries.
- Connect—Re-enables a link's address entries and connects to selected groups.
- Disconnect—Implicitly disables links and disconnects links to requested groups.

Inquire Function

The Inquire function of the link management request message allows querying of a single group's link state. To use the Inquire function, specify the group number of the local or remote group for which you want to learn the link state. This function does not allow you to specify any selection parameters other than the group number. Because you can only inquire about the link state of one group at a time, you cannot specify the `PSYM_LINKMGT_ALL_GROUPS` symbol in the group_number field.

The Inquire function performs endian translation when the request is sent to a Connect Server running on a system that uses a different byte order. Both the request and response messages are encoded in the endian of the request originator.

Request Message Format for the Inquire Function

Table 5-2 displays the Inquire function request message format:

Table 5-2 Inquire Function Request Message Format

Field	Required/ Optional	Setting
`version`	Required	10
`user_tag`	Required	User-specified code identifying the request.
`function_code`	Required	`PSYM_LINKMGT_CMD_INQUIRY`
`group_number`	Required	Group number to receive the action. Valid values are 1 to 32000.
`connect_type`	Optional	`PSYM_LINKMGT_ALL_TRANSPORTS`
`reconnect_timer`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`
`window_size`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`
`window_delay`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`

Table 5-2 Inquire Function Request Message Format

Field	Required/ Optional	Setting
`transport_addr_len`	Optional	0
`node_name_len`	Optional	0

Determining the Status of the Inquire Request

The status field of the `LINKMGT_RESP` message contains a return code indicating the outcome of the inquiry request. Refer to Table 5-3 for a description of each status return and the corresponding user action.

Table 5-3 Inquire function status returns and user actions

PSYM_LINKMGT Return Code	Description	Outcome	Description/User Action
`MSGCONTENT`	Invalid value in request message	Error	One of the field values in the inquiry request message is invalid. Check the syntax of the request message against the list of valid values and re-issue the corrected request message.
`MSGFMT`	Unknown request version or function code	Error	Correct the syntax of the request message. The version field of the must contain the number 10. The function code field must contain the symbol `PSYM_LINKMGT_CMD_INQUIRY`.
`NOGROUP`	The selected group does not have a cross group entry	Error	You requested the link state for a group that is not defined in the cross-group table. This group has no cross-group links.
`OPERATIONFAIL`	The command was unable to be successfully completed	Error	The inquire function failed due to a system resource problem. ■ Check the network connection to the target group to determine if the network link is up. ■ Check the Connect Server to determine if it is running out of virtual memory. ■ Check the log file to see if the cause of the error has been logged.

Table 5-3 Inquire function status returns and user actions

PSYM_LINKMGT Return Code	Description	Outcome	Description/User Action
`SUCCESS`	The operation successfully completed	Success	Refer to the description of the link management response message below for a description of the data returned.

Response Message Format for Successful Inquire Requests

If the Inquire function is successful, the response message returns the status of both the incoming and outgoing cross-group links in the in_link_state and out_link_state fields. These fields specify the status of the link using the following symbols:

- `PSYM_LINKMGT_CONNECTED`—the incoming/outgoing cross-group link for the selected group is connected.
- `PSYM_LINKMGT_NOCN`—the incoming/outgoing cross-group link for the selected group is not connected.
- `PSYM_LINKMGT_DISABLE`—the incoming/outgoing cross-group link for the selected group is disabled.

If the link status for the group is `PSYM_LINKMGT_CONNECTED`, the response message contains the following information:

Field	Description
`version`	10
`user_tag`	User-specified code from the request message.
`Status`	`PSYM_LINKMGT_SUCCESS`
`group_number`	Group number that receives the action.
`in_link_state`	`PSYM_LINKMGT_CONNECTED`
`out_link_state`	`PSYM_LINKMGT_CONNECTED`
`connect_type`	Transport that message is connected over: `PSYM_LINKMGT_TCPIP`.

Field	Description
`platform_id`	Connected platform ID (`PSYM_PLATFORM_`*`xxxx`*).
`reconnect_time`	Reconnect timer value for this group.
`window_size`	Window size value negotiated for this group.
`window_delay`	Window delay value negotiated for this group.
`transport_addr_len`	Length of the transport_addr string.
`transport_addr`	ASCII representation of the TCP/IP port number.
`node_name_len`	Length of the node_name string.
`node_name`	Name of the node this link is connected to.

Enable Function

The Enable function of the link management request message re-enables a link's address entries if they have been disabled. All addresses in the cross-group connection table that match the selection criteria specified in the request message (for example, group number, connect type, node name, and transport address) will be enabled. All other address entries for the group or groups selected will be disabled. The Enable function will still complete if the link is already connected. The effects will not be visible until the existing link is lost.

The Enable function allows a link to occur only with the selected addresses for a group. If the group has a reconnection timer, the timer will be set to cause the connection to be attempted after the specified time and connections are not attempted immediately. Incoming connections are then allowed to occur.

- The Enable function offers the following selection options:
- If the group_number field is set to `PSYM_LINKMGT_ALL_GROUPS`, then the node name and transport address cannot be specified.
- If a specific group number is specified and `PSYM_LINKMGT_ALL_TRANSPORTS` is specified, then the node name and transport address cannot be specified.
- On OpenVMS systems, if an entry that matches the selection criteria is not found, one will be created providing the group exists. On UNIX and Windows

NT systems, the Enable function only enables existing address entries. It does not modify connection parameters or add new address entries.

- On OpenVMS systems, if the window or reconnect timer information is supplied, the specified values overwrite the existing information of the select entries. On UNIX and Windows NT systems, the Enable function does not modify connection parameters.

Note: The symbol `PSYM_LINKMGT_ALL_TRANSPORTS` is new to the `LINK_MGT` message API for BEA MessageQ Version 4.0. On OpenVMS systems, the Enable function requires that the requesting process have either `OPER` or the `DMQ$OPERATOR` rights identifier.

Request Message Format for the Enable Function

Table 5-4 displays the Enable function message format:

Table 5-4 Enable function message format

Field	Required/ Optional	Setting
`version`	Required	10
`user_tag`	Required	User-specified code identifying the request.
`function_code`	Required	`PSYM_LINKMGT_CMD_ENABLE`
`group_number`	Required	Group number to receive the action. Valid values are 1 to 32000. Or, use the `PSYM_LINKMGT_ALL_GROUPS` symbol to enable all known links for groups with the connect_type requested.
`connect_type`	Required	Select the following transport type: `PSYM_LINKMGT_TCPIP`

`reconnect_timer`	Optional	Time it takes for the COM Server or Group Control Process (GCP) to reconnect to a communications link. Enter the number of seconds or the following values:
		`PSYM_LINKMGT_NO_TIMER`
		`PSYM_LINKMGT_USE_PREVIOUS`

Table 5-4 Enable function message format

Field	Required/ Optional	Setting
`window_size`	Optional	Size of transmission window (cross-group protocol Version 3.0 or higher).
`window_delay`	Optional	Transmission window delay in seconds (cross-group protocol Version 3.0 or higher).
`transport_addr`	Optional	Transport address string 16 bytes in length; the TCP/IP port ID
`transport_addr_len`	Optional	Length of transport address. Valid values are 0 to 16 bytes. Zero specifies the use of the previous setting.
`node_name`	Optional	ASCII text of node name. The length is determined by `node_name_len` up to 255 characters.
`node_name_len`	Optional	Length of the node name string. Zero specifies the use of the previous known value.

Determining the Status of the Enable Request

The status field of the `LINKMGT_RESP` message contains a return code indicating the outcome of the Enable request. See Table 5-5 for a description of each status return and the corresponding user action.

Table 5-5 Enable function status returns and user actions

PSYM_LINKMGT Return Code	Description	Outcome	Description/User Action
`ALREADYUP`	The link is already active	Warning	The Enable function completed although the link entries were already available.
`MSGCONTENT`	Invalid value in request message	Error	One of the field values in the enable request message is invalid. Check the syntax of the request message against the list of valid values and re-issue the corrected request message.

Table 5-5 Enable function status returns and user actions

PSYM_LINKMGT Return Code	Description	Outcome	Description/User Action
`MSGFMT`	Unknown request version or function code	Error	Correct the syntax of the request message. The version field of the must contain the number 10. The function code field must contain the symbol `PSYM_LINKMGT_CMD_ENABLE`.
`NOGROUP`	The selected group does not have a cross group entry	Error	No cross-group entries can be enabled because you requested the enable function for a group that is not defined in the cross-group table.
`NOTRANSPORT`	The selected group does not have any cross-group entries with specified transport	Error	No cross-group entries can be enabled because you requested the enable function for a group or groups that does not have a cross-group connection entry that uses the specified transport.
`OPERATIONFAIL`	The command was unable to be successfully completed	Error	The enable function failed due to a system resource problem. ■ Check the Connect Server to determine if it is running out of virtual memory. ■ Check the log file to see if the cause of the error has been logged.
`SUCCESS`	The operation successfully completed.	Success	Refer to the description of the link management response message below for a description of the data returned.

Response Message Format for Successful Enable Requests

If the Enable function is successful, the response message returns the information shown in the following table:

Field	Description
`version`	10
`user_tag`	User-specified code from the request message.
`status`	`PAMS__SUCCESS`
`group_number`	Group number or numbers to receive the action.
`in_link_state`	`PSYM_LINKMGT_ENABLED`
`out_link_state`	`PSYM_LINKMGT_ENABLED`
`connect_type`	Transport that message is connected over: `PSYM_LINKMGT_TCPIP`.
`platform_id`	Connected platform ID (`PSYM_PLATFORM_`*`xxxx`*).
`reconnect_time`	Reconnect timer value for this group.
`window_size`	Window size value negotiated for this group.
`window_delay`	Window delay value negotiated for this group.
`transport_addr_len`	Length of the transport_addr string.
`transport_addr`	ASCII representation of either the TCP/IP port number.
`node_name_len`	Length of the node_name string.
`node_name`	Name of the node this link is connected to.

Disable Function

The Disable function of the link management request message disables a link's address entries if they have been enabled. This prevents a link from occurring with the group's selected addresses. Connection attempts to and from the selected addresses are prevented.

All addresses in the group address table that match the selection criteria of the message (for example, group ID, connect type, node name, and transport address) will be disabled. All other address entries for the groups selected will not be affected. If no entry matches the group_number field, then `PSYM_LINKMGT_NOGROUP` is returned.

The Disable function takes matching cross-group entries out of the search list for connect processing.

Request Message Format for the Disable Function

Table 5-6 displays the Disable function message format:

Table 5-6 Disable Function Message Format

Field	Required/ Optional	Setting
`version`	Required	10
`user_tag`	Required	User-specified code identifying the request.
`function_code`	Required	`PSYM_LINKMGT_CMD_DISABLE`
`group_number`	Required	Group number to receive the action. Valid values are 1 to 32000. The `PSYM_LINKMGT_ALL_GROUPS` symbol indicates all known links for this group.
`connect_type`	Required	Select the following transport type: `PSYM_LINKMGT_TCPIP`
`reconnect_timer`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`
`window_size`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`
`window_delay`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`
`transport_addr`	Optional	Transport address string 16 bytes in length; the TCP/IP port ID
`transport_addr_len`	Optional	Length of transport address. Valid values are 0 to 16 bytes. Zero indicates to use the previous setting.
`node_name`	Optional	ASCII text of node name. The length is determined by node_name_len up to 255 characters.
`node_name_len`	Optional	Length of the node name string. Zero indicates to use the previous known value.

Determining the Status of the Disable Request

The status field of the `LINKMGT_RESP` message contains a return code indicating the outcome of the Disable request. See Table 5-7 for a description of each status return and the corresponding user action.

Table 5-7 Disable Function Status Return and User Action

PSYM_LINKMGT Return Code	Description	Outcome	Description/User Action
`MSGCONTENT`	Invalid value in request message	Error	One of the field values in the disable request message is invalid. Check the syntax of the request message against the list of valid values and re-issue the corrected request message.
`MSGFMT`	Unknown request version or function code	Error	Correct the syntax of the request message. The version field of the must contain the number 10. The function code field must contain the symbol `PSYM_LINKMGT_CMD_DISABLE`.
`NOGROUP`	The selected group does not have a cross group entry	Error	No cross-group entries can be disabled because you requested the disable function for a group that is not defined in the cross-group table.
`NOTRANSPORT`	The selected group does not have any cross group entries with specified transport	Error	No cross-group entries can be disabled because you requested the disable function for a group or groups that does not have a cross-group connection entry that uses the specified transport.
`OPERATIONFAIL`	The command was unable to be successfully completed	Error	The disable function failed due to a system resource problem. ■ Check the Connect Server to determine if it is running out of virtual memory. ■ Check the log file to see if the cause of the error has been logged.
`SUCCESS`	The operation successfully completed.	Success	Refer to the description of the link management response message below for a description of the data returned.

Response Message Format for Successful Disable Requests

If the Disable function completes successfully, the response message contains the following information:

Field	Description
`version`	10
`user_tag`	User-specified code from the request message.
`status`	`PSYM_LINKMGT_SUCCESS`
`group_number`	Group number that receives the action.
`in_link_state`	`PSYM_LINKMGT_DISABLED`
`out_link_state`	`PSYM_LINKMGT_DISABLED`
`connect_type`	Transport that message is connected over: `PSYM_LINKMGT_TCPIP`.
`platform_id`	Connected platform ID (`PSYM_PLATFORM_`*`xxxx`*).
`reconnect_time`	Reconnect timer value for this group.
`window_size`	Window size value negotiated for this group.
`window_delay`	Window delay value negotiated for this group.
`transport_addr_len`	Length of the transport_addr string.
`transport_addr`	ASCII representation of either the TCP/IP port number.
`node_name_len`	Length of the node_name string.
`node_name`	Name of the node this link is connected to.

Connect Function

The Connect function of the link management request message re-enables a link's address entries if they have been disabled, and causes an immediate connect attempt to occur with the selected groups if not already connected. Incoming connections are then allowed to occur. This function will still be able to complete even if the link is already connected. The effects of the function will not be visible until the existing link is lost.

All addresses in the group address table that match the selection criteria of the message (for example, group ID, connect type, node name, and transport address) will be enabled, and all other address entries for the groups selected will be disabled. If a matching entry is not found, then one will be created, providing the group exists. If the window or reconnect timer information is supplied, then those values will overwrite the existing information of the selected entries.

If the group_number field is set to `PSYM_LINKMGT_ALL_GROUPS`, then node name and transport address cannot be specified. If a specific group number is specified, and `PSYM_LINKMGT_ALL_TRANSPORTS` is specified, then node name and transport address cannot be specified.

On OpenVMS systems, the Connect function requires that the requesting process have either `OPER` or the `DMQ$OPERATOR` rights identifier.

Request Message Format for the Connect Function

Table 5-8 displays the Connect request function message format:

Table 5-8 Connect Request Function Message Format

Field	Required/ Optional	Setting
`version`	Required	10
`user_tag`	Required	User-specified code identifying the request, if supplied.
`function_code`	Required	`PSYM_LINKMGT_CMD_CONNECT`
`group_number`	Required	Group number to receive the action. Valid values are 1 to 32000. The `PSYM_LINKMGT_ALL_GROUPS` symbol indicates all known links for this group.
`connect_type`	Required	Select the following transport type: `PSYM_LINKMGT_TCPIP`
`reconnect_timer`	Optional	Time it takes for the COM Server to reconnect to a communications link. Enter the number of seconds or the following values: `PSYM_LINKMGT_NO_TIMER` `PSYM_LINKMGT_USE_PREVIOUS`

Table 5-8 Connect Request Function Message Format

Field	Required/ Optional	Setting
`window_size`	Optional	Size of transmission window (cross-group protocol Version 3.0 or higher).
`window_delay`	Optional	Transmission window delay in seconds (cross-group protocol Version 3.0 or higher).
`transport_addr`	Optional	Transport address string 16 bytes in length' the TCP/IP port ID
`transport_addr_len`	Optional	Length of transport address. Valid values are 0 to 16 bytes. Zero specifies the use of the previous setting.
`node_name`	Optional	ASCII text of node name. The length is determined by `node_name_len` up to 255 characters.
`node_name_len`	Optional	Length of the node name string. Zero specifies the use of the previous known value.

Determining the Status of the Connect Request

The status field of the `LINKMGT_RESP` message contains a return code indicating the outcome of the Connect request. See Table 5-9 for a description of each status return and the corresponding user action.

Table 5-9 Connect function status returns and user actions

PSYM_LINKMGT Return Code	Description	Outcome	Description/User Action
`ALREADYUP`	The link is already active	Warning	The Connect function completed although the link entries were already available.
`MSGCONTENT`	Invalid value in request message	Error	One of the field values in the connect request message is invalid. Check the syntax of the request message against the list of valid values and re-issue the corrected request message.

Table 5-9 Connect function status returns and user actions

PSYM_LINKMGT Return Code	Description	Outcome	Description/User Action
`MSGFMT`	Unknown request version or function code	Error	Correct the syntax of the request message. The version field of the must contain the number 10. The function code field must contain the symbol `PSYM_LINKMGT_CMD_CONNECT`.
`NOGROUP`	The selected group does not have a cross group entry	Error	No cross-group links can be connected because you requested the connect function for a group that is not defined in the cross-group table.
`NOTRANSPORT`	The selected group does not have any cross group entries with specified transport	Error	No cross-group links can be connected because you requested the connect function for a group or groups that does not have a cross-group connection entry using the specified transport.
`OPERATIONFAIL`	The command was unable to be successfully completed	Error	The connect function failed due to a system resource problem. ■ Check the Connect Server to determine if it is running out of virtual memory. ■ Check the log file to see if the cause of the error has been logged.
`SUCCESS`	The operation successfully completed.	Success	Refer to the description of the link management response message below for a description of the data returned.

Response Message Format for Successful Connect Requests

If the Connect request is successful, the response message contains the following information:

Field	Description
`version`	10
`user_tag`	User-specified code from the request message.
`status`	`PSYM_LINKMGT_SUCCESS`
`group_number`	Group number that receives the action.

Field	Description
`in_link_state`	`PSYM_LINKMGT_CONNECTED`
`out_link_state`	`PSYM_LINKMGT_CONNECTED`
`connect_type`	Transport that message is connected over: `PSYM_LINKMGT_TCPIP`.
`platform_id`	Connected platform ID (`PSYM_PLATFORM_`*xxxx*).
`reconnect_time`	Reconnect timer value for this group.
`window_size`	Window size value negotiated for this group.
`window_delay`	Window delay value negotiated for this group.
`transport_addr_len`	Length of the transport_addr string.
`transport_addr`	ASCII representation of either the TCP/IP port number.
`node_name_len`	Length of the node_name string.
`node_name`	Name of the node this link is connected to.

Disconnect Function

The Disconnect function of the link management request message requests implicit disables of links and disconnects any links to the requested group. All addresses in the group address table that match the selection criteria of the message (for example, group ID, connect type, node name, and transport address) will be disconnected. All other address entries for the groups selected will not be affected. If no entry matches the group_number field, then `PSYM_LINKMGT_NOGROUP` is returned. On OpenVMS systems, the Disconnect function requires that the requesting process have either `OPER` or the `DMQ$OPERATOR` rights identifier.

Request Message Format for the Disconnect Function

Table 5-10 displays the Disconnect function message format.

Table 5-10 Disconnect Function Message Format

Field	Required/ Optional	Setting
`version`	Required	10
`user_tag`	Required	User-specified code identifying the request.

Table 5-10 Disconnect Function Message Format

Field	Required/ Optional	Setting
`function_code`	Required	`PSYM_LINKMGT_CMD_DISCONNECT`
`group_number`	Required	Group number to receive the action. Valid values are 1 to 32000. The `PSYM_LINKMGT_ALL_GROUPS` symbol means disconnect all known links for this group.
`connect_type`	Required	Select the following transport type: `PSYM_LINKMGT_TCPIP`
`reconnect_timer`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`
`window_size`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`
`window_delay`	Optional	`PSYM_LINKMGT_USE_PREVIOUS`
`transport_addr`	Optional	Transport address string 16 bytes in length; the TCP/IP port ID
`transport_addr_len`	Optional	Length of transport address. Valid values are 0 to 16 bytes. Zero specifies the use of the previous setting.
`node_name`	Optional	ASCII text of node name. The length is determined by `node_name_len` up to 255 characters.
`node_name_len`	Optional	Length of the node name string. Zero specifies the use of the previous known value.

Determining the Status of the Disconnect Request

The status field of the `LINKMGT_RESP` message contains a return code indicating the outcome of the Disconnect request. Refer to Table 5-11 for a description of each status return and the corresponding user action.

Table 5-11 Disconnect function status returns and user actions

PSYM_LINKMGT Return Code	Description	Outcome	Description/User Action
`MSGCONTENT`	Invalid value in request message	Error	One of the field values in the disconnect request message is invalid. Check the syntax of the request message against the list of valid values and re-issue the corrected request message.
`MSGFMT`	Unknown request version or function code	Error	Correct the syntax of the request message. The version field must contain the number 10. The function code field must contain the symbol `PSYM_LINKMGT_CMD_DISCONNECT`.
`NOGROUP`	The selected group does not have a cross-group entry	Error	No cross-group connections can be disconnected because you requested the disconnect function for a group that is not defined in the cross-group table.
`NOTRANSPORT`	The selected group does not have any cross-group entries with specified transport	Error	No cross-group links can be disconnected because you requested the disconnect function for a group or groups that does not have a cross-group connection entry that uses the specified transport.
`OPERATIONFAIL`	The command was unable to be successfully completed	Error	The disconnect function failed due to a system resource problem. ■ Check the Connect Server to determine if it is running out of virtual memory. ■ Check the log file to see if the cause of the error has been logged.
`SUCCESS`	The operation successfully completed.	Success	Refer to the description of the link management response message below for a description of the data returned.

Response Message Format for Successful Disconnect Functions

If the Disconnect function is successful, the response message returns the following information:

Field	Description
version	10
user_tag	User-specified code from the request message.
status	PSYM_LINKMGT_SUCCESS
group_number	Group number that receives the action.
in_link_state	PSYM_LINKMGT_DISABLED
out_link_state	PSYM_LINKMGT_DISABLED
connect_type	Transport that message is connected over: PSYM_LINKMGT_TCPIP.
platform_id	Connected platform ID (PSYM_PLATFORM_*xxxx*).
reconnect_time	Reconnect timer value for this group.
window_size	Window size value negotiated for this group.
window_delay	Window delay value negotiated for this group.
transport_addr_len	Length of the transport_addr string.
transport_addr	ASCII representation of either the TCP/IP port number.
node_name_len	Length of the node_name string.
node_name	Name of the node this link is connected to.

Link Management Design Considerations

Table 5-12 lists important design considerations for applications using link management.

Table 5-12 Link Management Design Condsiderations

Feature	Description
Failover Node Table Disabled	When an application issues a `LINKMGT_REQ` request, the Connect Server disables the failover node table defined in the group initialization file. Disabling the failover node table ensures the application complete control over the attributes of the link request.
Additional Group Connections Disabled	When the application issues a `LINKMGT_REQ` request to disconnect a link, the Connect Server disables further connections to the group. Disabling connections ensures that no additional links to the group will occur until the application issues another `LINKMGT_REQ` request.
Connect Requests Verified	When a connect request is made for a single group, the `XGROUP_VERIFY` table uses the information supplied in the message to determine whether to accept or reject the request for a connection. Cross-group verification only works on incoming requests. The data structure for cross-group verification is overwritten by the information in the link management connect or disconnect message.
Connect and Disconnect Requests Acknowledged	When the Connect Server receives a connect message after a link is already successfully connected, the Connect Server rejects the second connect message. When the Connect Server receives a disconnect message after a link is already successfully disconnected, the Connect Server acknowledges the second disconnect message with a successful return message.
Restrictions on Local and Remote Requests	The Connect Server will only accept link control requests from a local application. However, the Connect Server will accept link status inquiries from remote as well as local applications.
Privileges Required	Application link control requests on the OpenVMS system require that the application have VMS `OPER` privilege or be granted the `DMQ$OPERATOR` rights identifier.

Learning the Current Status of Queues

This section describes how applications can use Queue Server message-based services to obtain status information on all active queues in a particular group or to obtain notification of queue status changes. The list of active queues displays all attached permanent and temporary queues.

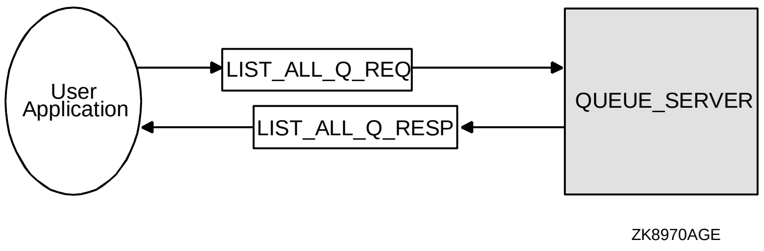
Listing Attached Queues in a Group

The Queue Server process can provide applications with a list of all attached queues for a selected group. This information is available for local and remote groups and includes a listing of both permanent and temporary queues. To request this list, the application program sends a message of type `LIST_ALL_Q_REQ` to the Queue Server process.

To learn the status of all queues in a selected group, an application exchanges the following messages with the Queue Server:

- `LIST_ALL_Q_REQ`—Request message to request the status of all queues.
- `LIST_ALL_Q_RESP`—Response message to provide a list of all queues and their status.

Figure 5-5 Listing All Queues



The application receives a response message from the Queue Server of type `LIST_ALL_Q_RESP` providing a list of all attached queues. Because a `LIST_ALL_Q_RESP` message may contain a long list of queue names, the application must allocate a sufficient buffer size to store the information returned.

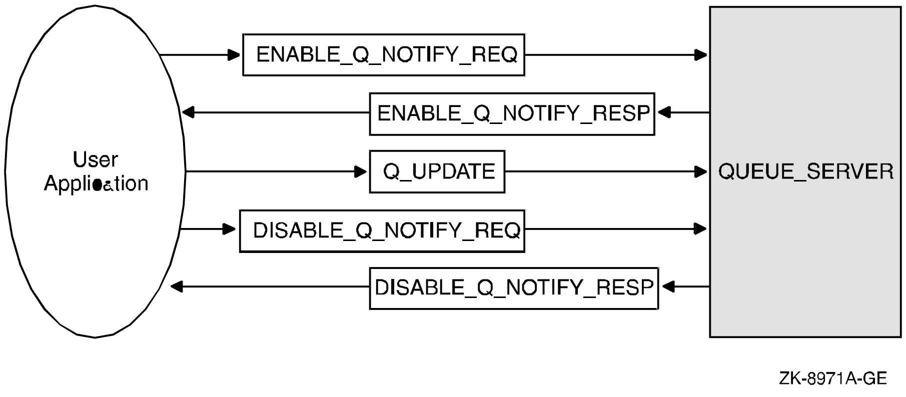
Receiving Attachment Notifications

The Queue Server process can notify an application of all attached queues and subsequent queue attachments and detachments for its own group. An application registers for this service by sending a message of type `ENABLE_Q_NOTIFY_REQ` to the group's Queue Server process. The Queue Server responds with a message of type `ENABLE_Q_NOTIFY_RESP`, indicating the status of the registration request.

To learn the status of all queues and receive ongoing notification of new queue attachments and detachments, an application exchanges the following messages with the Queue Server:

- `ENABLE_Q_NOTIFY_REQ`—Request message to request the current status of all queues with notification of future queue status changes.
- `ENABLE_Q_NOTIFY_RESP`—Response message to provide the current status of all queues and confirmation that queue status changes will be reported.
- `Q_UPDATE`—Notification message to provide information on newly attached and detached queues in the selected group.
- `DISABLE_Q_NOTIFY_REQ`—Request message to request that notification of queue status changes be discontinued.
- `DISABLE_Q_NOTIFY_RESP`—Response message to indicate that notification of queue status changes has been successfully disabled.

Figure 5-6 Listing Available Queues



The registration request places the sender’s response queue number in the list of applications to receive notification of new attachments and detachments. Notifications are sent using a message of type `Q_UPDATE`. The application can cancel the notification registration by sending a message of type `DISABLE_Q_NOTIFY_REQ`. The Queue Server responds with a reply of type `DISABLE_Q_NOTIFY_RESP` indicating the status of the registration cancellation request.

Managing Message Recovery Files

BEA MessageQ message-based services are used with the MRS Server to maintain files for recoverable messaging and to turn MRS journaling capability on or off. Message-based services for performing these functions are available on OpenVMS systems only. The functions are also available through the BEA MessageQ Manager Utility on OpenVMS systems. For complete information on how to use the BEA MessageQ message recovery system, see the Sending Recoverable Messages topic.

BEA MessageQ uses the following four BEA MessageQ files for MRS message-based services:

Store and forward file (SAF)	Messages designated for recovery on the sender system.
Destination queue file (DQF)	Messages designated for recovery on the receiver system.
Dead letter journal (DLJ)	Undelivered messages not designated for recovery by BEA MessageQ. These messages can be delivered later from the DLJ by an application program.
Postconfirmation journal (PCJ)	Successfully delivered recoverable messages which form an audit trail of messaging events.

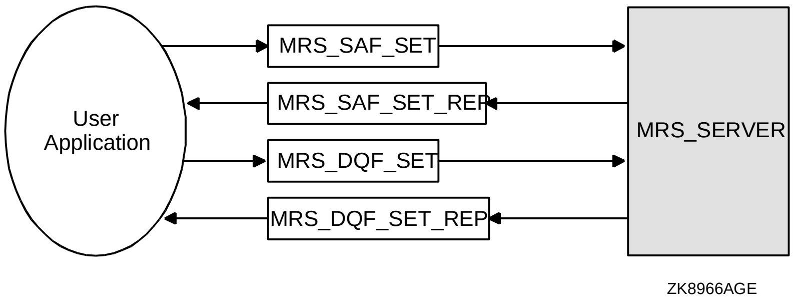
Opening, Closing, and Failing Over SAF and DQF Files

As part of message recovery on OpenVMS systems, the MRS Server opens a SAF or DQF file when a recoverable message is sent to the target queue. The following BEA MessageQ MRS message-based services are used to open, close, or rename message recovery files on Open VMS systems:

- `MRS_SAF_SET`—Request message to request the MRS Server to open, close, or rename the SAF file.
- `MRS_SAF_SET_REP`—Response message to indicate the status of the request.
- `MRS_DQF_SET`—Request message to request the MRS Server to open, close, or rename the file.
- `MRS_DQF_SET_REP`—Response message to indicate the status of the request.

Figure 5-7, MRS Server Message Flow, describes how to open, close, or rename message recovery files on OpenVMS systems:

Figure 5-7 MRS Server Message Flow



The `MRS_DQF_SET` message can be used to explicitly control the opening and closing of a DQF file. For example, an OpenVMS application can use the `MRS_DQF_SET` message to open a DQF file created at runtime in order to adjust its size before it begins storing recoverable messages. The `MRS_SAF_SET` message performs the same function for the SAF file.

The failover option of the `MRS_DQF_SET` message renames a DQF file, associating it with a new target queue that does not have a DQF file. The failover operation renames the destination queue file, and the messages in the store and forward (SAF) files directed to the original target are forwarded to the new target queue. You can use the `MRS_SAF_SET` message to fail over the SAF file.

The `MRS_SAF_SET_REP` and the `MRS_DQF_SET_REP` messages are responses to a request to open, close, or fail over an SAF or DQF file. The response message provides the status of the request.

Opening and Closing Auxiliary Journal Files

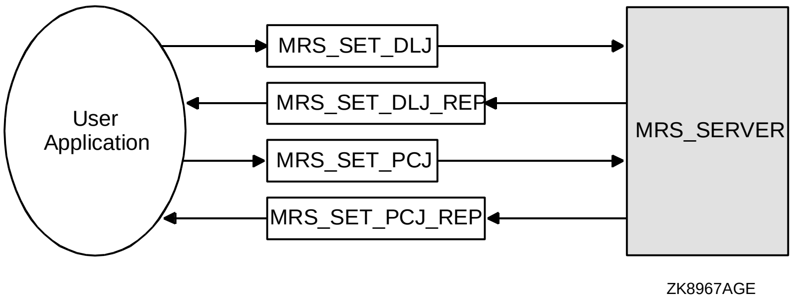
The dead letter journal (DLJ) file cannot be open simultaneously for read and write operations. For this reason, if an application has the task of delivering messages written to the DLJ file, it must close the current file before it can begin delivering the messages collected in it. The application must also open a new DLJ file to continue collecting undeliverable messages while it delivers the messages.

To open, close, or rename message recovery files on OpenVMS systems, an application exchanges the following messages with the MRS Server:

- `MRS_SET_DLJ`—Request message to request that the current DLJ file be closed and a new one opened.
- `MRS_SET_DLJ_REP`—Response message to indicate the status of the request.
- `MRS_SET_PCJ`—Request message to request that the current PCJ be closed and a new one opened.
- `MRS_SET_PCJ_REP`—Response message to indicate the status of the request.

Figure 5-8 illustrates the message exchange between the application and the MRS Server.

Figure 5-8 MRS Server Message Flow



The `MRS_SET_DLJ` message requests the MRS Server to close the current DLJ file and open a new one. The response message `MRS_SET_DLJ_REP` returns the status of the operation. The file specification for the newly created DLJ file is returned if the file is successfully opened.

As with the DLJ file, the postconfirmation journal (PCJ) file cannot be open simultaneously for read and write operations. For this reason, if an application has the task of reading the PCJ file to write a report of successful messaging transactions, it must first close the current file. The application must also open a new PCJ file to continue collecting successfully delivered recoverable messages.

An `MRS_SET_PCJ` message requests the MRS Server to close the current PCJ file and open a new one. An `MRS_SET_PCJ_REP` message is returned to the requestor and includes the status of the operation and the file specification if the file was successfully opened.

Note: In contrast to the MRS messages for opening and closing the SAF and DQF files, the DLJ and PCJ auxiliary recovery journals can be opened and closed in a single operation.

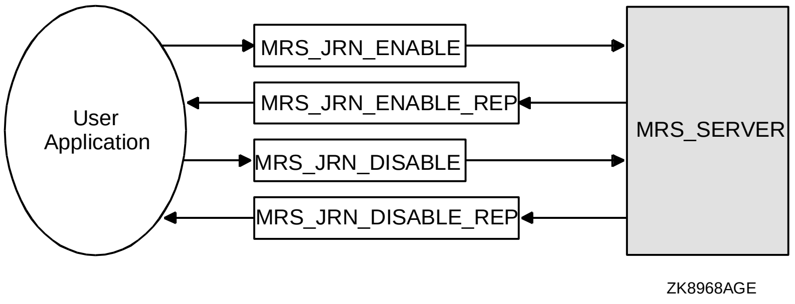
Controlling Journaling to the PCJ File

You can use the messages in Figure 5-9 to disable journaling when replacing a PCJ file and then reenable journaling:

- `MRS_JRN_DISABLE`—Request message to disable journaling to the PCJ file.

- MRS_JRN_DISABLE_REP—Response message to indicate the status of the request.
- MRS_JRN_ENABLE—Request message to enable journaling to the PCJ file.
- MRS_JRN_ENABLE_REP—Response message to indicate the status of the request.

Figure 5-9 Disabling Journaling



Use the `MRS_JRN_DISABLE` message to disable journaling to the PCJ when you need to close the PCJ and open a new one. The `MRS_JRN_DISABLE_REP` message returns the status of the operation. Use the `MRS_JRN_ENABLE` message to enable journaling after you have opened a new PCJ file. The `MRS_JRN_ENABLE_REP` message returns the status of the operation.

Transferring the Contents of a Destination Queue File

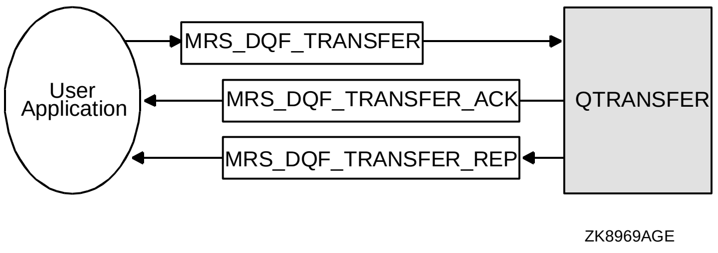
You use MRS Server messages to transfer the entire contents of a DQF file at once. However, you can use Qtransfer Server messages to request that the contents of a DQF file be transferred one message at a time into another DQF file.

Transferring the contents of one DQF to another queue supports a failover scheme allowing an application on a node that is running to process messages from a DQF file on a node that is not running. Using Qtransfer Server messages you can blend the contents of two DQFs. Qtransfer Server messages are available on OpenVMS systems only.

To transfer the contents of one DQF file to another, an application exchanges the following messages with the Qtransfer Server:

- `MRS_DQF_TRANSFER`—Request message to request the transfer of the contents of a DQF file to another.
- `MRS_DQF_TRANSFER_ACK`—Notification message to acknowledge receipt of the request.
- `MRS_DQF_TRANSFER_REP`—Response message to indicate the final status of the transfer.

Figure 5-10 Qtransfer Server Message Flow



The `MRS_DQF_TRANSFER` message requests the Qtransfer Server to open a DQF file and send its contents one message at a time to another recoverable queue. By providing a method for blending two recoverable queues, the Qtransfer Server provides a convenient failover mechanism when application processing is conducted on multiple nodes in a distributed processing network.

Using this failover method, when a node fails, Qtransfer Server messages can be used to transfer messages from a recoverable queue on a node that has failed to a recoverable queue on a node that is currently processing messages.

To acknowledge the receipt of an `MRS_DQF_TRANSFER` message, the Qtransfer Server sends an `MRS_DQF_TRANSFER_ACK` message. When each message is successfully stored in the target DQF file, it is deleted from the source DQF file.

When all messages have been successfully stored in the target DQF file, or if an error has stopped the transfer, a message of type `MRS_DQF_TRANSFER_REP` is sent. The `MRS_DQF_TRANSFER_REP` message indicates the completion status of a message of type `MRS_DQF_TRANSFER`.

CHAPTER

6 Building and Testing Applications

This chapter describes the following tasks:

- Formatting and Converting Message Data
- Writing Portable BEA MessageQ Applications
- Compiling and Linking BEA MessageQ Applications
- Using the BEA MessageQ Test Utility
- Debugging BEA MessageQ Applications
- Controlling Message Flow

Formatting and Converting Message Data

Computer systems from different manufacturers may format data and data structures differently. When sending messages between computers in a multivendor environment, the process of **data marshaling** ensures that data is interpreted properly between the sending and receiving systems.

The FML-based self-describing messaging feature in BEA MessageQ allows applications to construct messages that contain information about how to interpret the message content. Therefore, FML performs data marshaling to handle byte order and data alignment differences between computer systems. See the Self-Describing Messaging topic for more information.

Byte Order Conversion

Computer systems use two different methods to store a single integer value as a longword. A longword, which represents 4 bytes, can be stored from highest to lowest address order or from lowest to highest.

The term **endian** refers to the end of the longword that the computer begins reading first. Some computers read the longword beginning with the lowest byte address, a format called little endian. Other computers read the longword starting with the highest byte address, a format called big endian.

When information is exchanged between computer systems that use different endian formats, the format must be agreed upon by the two systems. Otherwise, the target system will read the data and interpret the wrong integer value. The sender program can convert the data by reversing the order of the bytes before the data is sent over the network. Or the receiver program can reverse the order of the bytes before it interprets the integer.

The convention for sending data between dissimilar endian machines is to use network byte order (big endian format) to pack data into a message before sending it. The show buffer argument of the `pams_get_msg` function returns the endian format of the system that originated the message. The endian field is applicable to an integer-only format.

One way to avoid having to perform endian conversion is to convert numbers into character strings and to *only* use messages composed entirely of ASCII text data. While requiring more buffer space, text-only messages are always completely portable for reception on any system.

If you must format message data using numeric data types, you can use several conversion functions to convert the network byte order of the messages between systems that use different endian formats. Many systems or C compilers provide the `ntohl`(), `ntohs`(), `htonl`(), and `htons`() functions. These functions convert numeric data types to or from their host internal representation into a common standard format (called network byte order) for message transmission.

If these functions are not available, a user function could be written to produce the same results. Network byte order arranges the bytes with the most significant bits at the lower addresses.

Alignment of Data Structures

In addition to converting data between different endian formats, BEA MessageQ applications may also need to align message data structures to ensure that message content is interpreted properly by the target system. Many RISC compilers automatically perform data alignment during program compilation. When a program is compiled, data elements within a structure are aligned along natural boundaries by data type such as byte, word, or longword.

Alignment causes data elements to shift position when space is added to align the elements along these boundaries. Aligning data helps programs to run more efficiently. However, because elements are moved and space is added to the structure, alignment changes the way in which the data structure must be read.

Developers can use one of the following methods to ensure that message data structures are not changed by data alignment:

- Suppress data alignment during compilation. Many compilers allow developers to set a switch that enables and disables data alignment at compile time.
- Develop the application to create a packed data format for standard messages. If the application formats the data structure as a byte array, the packed data format is preserved during compilation, even when using compilers that automatically align data.

- Design the data structure so that the elements in it are naturally aligned. Natural alignment ensures that all longwords are on 4 byte boundaries, all words are on 2 byte boundaries, and so on.

See your system documentation for more information about data formatting on your system.

Writing Portable BEA MessageQ Applications

The best approach in developing BEA MessageQ applications is to use portable programming techniques that allow the application to run in many different computing environments. Writing portable applications reduces development and maintenance costs as applications are required to run on systems from many vendors.

The following suggestions for developing BEA MessageQ applications simplify porting applications to all industry-leading platforms:

- Avoid using nonportable BEA MessageQ features.
- Some BEA MessageQ functions (such as the `pams_get_msga` function) and other features are not available on all platforms. The PAMS API reference information lists which functions are not available on all platforms.
- Specify all optional arguments in BEA MessageQ functions.

Only OpenVMS systems allow applications to omit the trailing arguments in a function call if they are not required. All other BEA MessageQ implementations require that each argument in a BEA MessageQ function be specified. Arguments that are not required by the application should be specified by passing a value of zero.

Compiling and Linking BEA MessageQ Applications

This topic describes how to build your BEA MessageQ applications for UNIX, Windows NT, and OpenVMS environments. The following sections describe:

- Using BEA MessageQ Include Files
- Connecting to the BEA MessageQ Environment
- Compiling and Linking Applications
- Running a BEA MessageQ Application
- Testing Return Status

Using BEA MessageQ Include Files

To use BEA MessageQ API functions and other standard features in your application, reference the BEA MessageQ include files at the beginning of your application program. Table 6-1 describes the contents of each BEA MessageQ include file. The include files can be used with both the C and C++ programming languages.

Table 6-1 BEA MessageQ Include Files

File Name	Comments	Description
`p_entry.h`	Entry point definitions	Declares the entry point for all BEA MessageQ API calls.
`p_proces.h`	Process definitions	Defines the queue numbers symbolically to identify other queues in the BEA MessageQ message queuing system.
`p_group.h`	Group definitions	Defines the group numbers symbolically to identify other groups in the BEA MessageQ message queuing system.

Table 6-1 BEA MessageQ Include Files

File Name	Comments	Description
`p_typecl.h`	Type and class definitions	Contains the symbolic names for all standard BEA MessageQ type and class definitions. On OpenVMS systems you can add user-defined type and class codes to this file. On UNIX and Windows NT systems you must create a separate include file for user-defined type and class codes.
`p_return.h`	Return code definitions	Contains the compile time symbols for BEA MessageQ return status codes.
`p_symbol.h`	Global symbol definitions	Defines symbols used by BEA MessageQ to control features such as message selection and recoverable messaging.
`p_msg.h`	Message API definitions	Declares the data structures for all BEA MessageQ message-based services.

All implementations of BEA MessageQ software access the C language include files in the same manner. Listing 6-1 shows the recommended method of specifying portable `#include` statements in C.

Listing 6-1 Recommended #include Statements for BEA MessageQ Applications

```
#include <errno.h>
#include <stdio.h>
        /* Include PAMS-specific definition files.            */
#include <p_entry.h>     /* PAMS function type declarations   */
#include <p_proces.h>    /* Known Queue number definitions.   */
#include <p_group.h>     /* Known group number definitions.   */
#include <p_typecl.h>    /* Generic type/class definitions.   */
#include <p_return.h>    /* PAMS func return status definitions*/
#include <p_symbol.h>    /* Generic PSEL/PDEL definitions.    */
#include  <p_msg.h>       /* Message type declarations           */
```

Portable code requires a conditional compile (such as `#if/#endif` when programming in C) around the include statements. For an example of how to incorporate include files into your application, refer to the sample programs in the examples directory of your BEA MessageQ media kit.

To use BEA MessageQ functions and other standard features in an application program, the program references the BEA MessageQ include files. Table 6-2 lists the location of the standard BEA MessageQ include files for the C programming language.

Table 6-2 Location of C Language Include Files

Platform	Location
UNIX	`/usr/include` directory
Windows NT	directory selected during installation
OpenVMS	`DMQ$USER:`

BEA MessageQ for OpenVMS systems uses the portable include file names for the C programming language. For other programming languages, BEA MessageQ uses another set of names for the include files. The OpenVMS include files for all other languages are contained in a single library called `DMQ.TLB`. The logical name `DMQ$USER` points to the directory containing `DMQ.TLB`.

Include files on OpenVMS systems are available for several programming languages. The include files begin with PAMS_*XXX*_ where *XXX* is a 1- to 3-letter designation identifying the programming language as follows:

- PAMS_*XXX*_ENTRY_POINT
- PAMS_*XXX*_PROCESS
- PAMS_*XXX*_GROUP
- PAMS_*XXX*_TYPE_CLASS
- PAMS_*XXX*_SYMBOL_DEF

BEA MessageQ for OpenVMS systems uses two different include files for return code symbols. Compile-time symbols are contained in the PAMS_*XXX*_RETURN_STATUS_DEF file. Link-time symbols are contained in the PAMS_*XXX*_RETURN_STATUS file. Include one of the following in your application program:

- PAMS_*XXX*_RETURN_STATUS_DEF
- PAMS_*XXX*_RETURN_STATUS

Programming Language Support

Table 6-3 shows the languages supported by BEA MessageQ products:

Table 6-3 Languages Supported By BEA MessageQ

Product	Supported Languages
BEA MessageQ for UNIX BEA MessageQ UNIX Client	C, C++
BEA MessageQ for Windows NT BEA MessageQ Windows Client	C, C++, Visual Basic, Powerbuilder
BEA MessageQ for OpenVMS BEA MessageQ OpenVMS Client	Ada, Basic, Bliss-32, C, C++, Cobol, Fortran, Macro-32, PL/I, Pascal
BEA MessageQ MVS Client	C, Cobol

Connecting to the BEA MessageQ Environment

Before running a program that uses BEA MessageQ, you must set the environment to identify the message queuing bus and the message queuing group with which the program will be associated.

For UNIX and Windows NT, BEA MessageQ uses the following environment variables:

`DMQ_BUS_ID`	Sets the bus ID for the application.
`DMQ_GROUP_ID`	Sets the group ID for the application.

To set environment variables that designate bus and group ID using `csh` syntax on UNIX systems, enter the following commands:

```
#  setenv DMQ_BUS_ID bus_id
#  setenv DMQ_GROUP_ID group_id
```

To set environment variables that designate bus and group ID using command line syntax on Windows NT systems, enter the following commands:

```
set DMQ_BUS_ID bus_id
set DMQ_GROUP_ID group_id
```

BEA MessageQ for OpenVMS enables you to tailor your run-time environment using OpenVMS logical names. You can use the command `DMQ$SET_LNM_TABLE` to place the required logical name table into the user's logical name search tree. This command procedure requires two parameters: the bus ID and the group ID.

Enter the following command to execute this procedure:

```
$ @MY_DMQ_DISK:[DMQ$V50.EXE]DMQ$SET_LNM_TABLE bus_id group_id
```

If the user frequently uses a particular bus and group, the invocation of the command procedure can be added to the user's `LOGIN.COM` file. The system manager can define the symbol `DMQ_SET` to simplify this command procedure. See the *BEA MessageQ Configuration Guide for OpenVMS* for more information.

Table 6-4 describes logical names that are useful for testing, monitoring, and debugging operations. See the *BEA MessageQ Configuration Guide for OpenVMS* for a complete list of BEA MessageQ logical names.

Table 6-4 Logical Names Used in Testing and Debuggung

Logical Name	Description
`DMQ$DCL_NUMBER`	This logical name is a base 10 queue number that overrides the queue number submitted to `pams_attach_q`.
`DMQ$EXIT_PURGE`	This logical name controls the purging of pending messages when the program exits. When defined as `NO`, it disables the BEA MessageQ exit handler from purging all primary and secondary queues attached to the process. This feature has no effect on pending recoverable messages because they are always requeued when `pams_attach_q` is called to attach to the queue.

Table 6-4 Logical Names Used in Testing and Debuggung

Logical Name	Description
DMQ$DEBUG	Some special features are incorporated into BEA MessageQ to aid in debugging. The logical name DMQ$DEBUG can be set to one of the following states by using the DCL DEFINE command: ■ undefined—No special action. ■ NORMAL—No special action. ■ ERROR—Prints error messages to the local terminal whenever an error occurs in a call to a BEA MessageQ function. ■ TRACE—This is a superset of the ERROR state. When set it will print the occurrence of any errors within BEA MessageQ. It will also print an informational message whenever a BEA MessageQ routine is entered. ■ ALL—Combines the functions of ERROR and TRACE.
DMQ$HEADERS	This logical name controls the printing of BEA MessageQ headers on the SYS$OUTPUT device; for example, the terminal. When this logical name is defined, BEA MessageQ header information is displayed when messages are sent to or received from this process.
DMQ$TRACE_OUTPUT	This logical name defines the location where trace information is logged.

Compiling and Linking Applications

This topic describes how to compile and link your BEA MessageQ application on UNIX, Windows NT and OpenVMS systems. For BEA MessageQ V5.0, the default compilation lines are as follows:

```
cc -I$BEADIR/include file.c -L$BEADIR/lib -ldmq      (direct call)
cc -I$BEADIR/include file.c -L$BEADIR/lib -ldmqcl    (client/server)
```

For use with FML32, the compilation line is as follows:

```
cc -I$BEADIR/include file.c -L$BEADIR/lib -lfml32 -lgp -ldmq
(direct call)
cc -I$BEADIR/include file.c -L$BEADIR/lib -lfml32 -lgp -ldmqcl
(client/server)
```

$BEADIR/lib must be included in the exported environment variable LIBPATH on AIX, SHLIB_PATH on HPUX, and LD_LIBRARY_PATH on all other Unix platforms.

The following additional compilation flags and/or libraries are needed on the various platforms:

HPUX

Compilation using the BEA MessageQ header files requires `-D_HPUX_SOURCE - Aa`

SCO Open Server 5.0:

replace `-ldmq` with `-Bdynamic -ldmq -lsocket`
replace `-ldmqcl` with `-Bdynamic -ldmqcl -lsocket`

SCO UnixWare

`-lgp` (if used) needs `-lcrypt`
`-ldmqcl` needs `-lsocket`

Sequent

`-ldmqcl` needs `-lsocket -lnsl`

NCR

`-ldmqcl` needs `-lsocket -lnsl`

Solaris

`-ldmqcl` needs `-lsocket -lnsl`

The following sections describe:

- UNIX Makefile
- Windows NT Makefile
- OpenVMS Build Procedure

UNIX Makefile

UNIX systems use a makefile to incorporate the commands for compiling and linking application programs. Listing 6-2 shows a sample makefile for running a BEA MessageQ for UNIX application. The sample makefile, with the client and server programs, is included in the root directory of your BEA MessageQ for UNIX media kit.

Listing 6-2 UNIX Makefile

```
# library to link against libdmq.a in /usr/lib
```

```
LIBS   =   -ldmq

# compiler flags include debugging symbols,  don’t use prototypes

CFLAGS =   -g

# build both the client and the server

all:       s_client s_server
# client depends on s_client.o s_getopt.o

s_client:  s_client.o s_getopt.o
           cc $(CFLAGS) s_client.o s_getopt.o $(LIBS) -o s_client

# server depends on s_server.o s_getopt.o

s_server:  s_server.o s_getopt.o
           cc $(CFLAGS) s_server.o s_getopt.o $(LIBS) -o s_server
```

When building BEA MessageQ applications on the Compaq Tru64 UNIX platform, you must link your applications against the library `libots.a` in addition to the BEA MessageQ library. For example:

```
# cc myapp.c -ldmq -lots - myapp
```

Windows NT Makefile

The BEA MessageQ for Windows NT API is implemented in dynamic link libraries (DLLs). The directory containing the DLLs must be in your path when you run your applications. All BEA MessageQ for Windows NT API functions are exported by `DMQ.DLL` and defined in the import library `DMQ.LIB`. Other Windows NT products that can call DLLs can also call BEA MessageQ API functions.

BEA MessageQ for Windows NT systems provides full support for Windows NT multithreading. Each thread in the BEA MessageQ process has an independent BEA MessageQ context, which means a queue that is attached in one thread is not available to another thread in the same process.

All threads must attach their own queues via the `pams_attach_q` function. When a program thread issues a `pams_exit` call, it does not affect queues attached by other threads in the same process. Multiple threads in one application can communicate via BEA MessageQ exactly as though they were in separate processes.

The example Windows NT makefile, `x_make.mak,` provides a good starting point for building your own makefiles. To link with BEA MessageQ for Windows NT systems, you need only include `DMQ.LIB` when you link. The following example shows a sample makefile for running a BEA MessageQ for Windows NT application. Listing 6-3 makefile is included in the examples directory of your BEA MessageQ for Windows NT media kit.

Listing 6-3 Windows NT Makefile

```
#
# x_make.mak:     Example makefile for MessageQ applications.
#                 This makefile builds the "simple client" and
#                 "simple server" applications.
#
# execute this file with NMAKE as follows:
#
# NMAKE -f example.mak DMQ=drive:\dir\
#
# where "drive" is the drive where MessageQ is installed and
#     "directory" is the directory where MessageQ is installed.
#
#
!include <ntwin32.mak>

BIN=.\
DLIB=$(DMQ)
SRC= $(DMQ)
OBJ=.\
INCDIRS= /I $(DMQ)

I_FILES= p_return.h p_entry.h p_symbol.h

ALL : $(BIN)s_client.exe $(BIN)s_server.exe

$(BIN)s_client.exe : s_client.obj s_getopt.obj
    $(link) $(conflags) \
         -out:$(BIN)s_client.exe     \
         s_client.obj s_getopt.obj   \
         $(DLIB)dmq.lib \
         $(conlibs)

$(BIN)s_server.exe : s_server.obj s_getopt.obj
    $(link) $(conflags) \
         -out:$(BIN)s_server.exe     \
         s_server.obj s_getopt.obj   \
         $(DLIB)dmq.lib \
         $(conlibs)
```

```
s_server.obj : $(SRC)s_server.c $(I_FILES)
  $(cc) $(cflags) $(cvars) $(SRC)s_server.c $(INCDIRS)

s_client.obj : $(SRC)s_client.c $(I_FILES)
  $(cc) $(cflags) $(cvars) $(SRC)s_client.c $(INCDIRS)

s_getopt.obj : $(SRC)s_getopt.c $(I_FILES)
  $(cc) $(cflags) $(cvars) $(SRC)s_getopt.c $(INCDIRS)
```

OpenVMS Build Procedure

This topic describes how to compile and link BEA MessageQ applications on OpenVMS systems. The BEA MessageQ for OpenVMS media kit includes a sample command procedure for compiling and linking BEA MessageQ applications. Listing 6-4 shows the sample build file included in the examples directory.

Listing 6-4 Example OpenVMS Build Procedure

```
$!=================================================================
$!   Standardized examples build procedure (V1.0-00)
$!
$!   File: X_BUILD.COM
$!
$!   Params: none
$!=================================================================
$
$ ss$_badparam    =  20
$ ss$_nopriv      =  36
$ ss$_abort       =  44
$ cc_alpha        = "/stand=vaxc/debug/noopt/lis"
$ cc_alpha_strict = "/stand=relaxed/debug/noopt/lis"
$ cc_vax          = "/stand=portable/debug/noopt/lis"
$ wl              = "write sys$output"
$
$
$ on warning then exit 4
$ on control_y then exit ’ss$_abort’
$
$ a = f$edit(f$getsyi("ARCH_NAME"), "UPCASE")
$ if f$locate("’’a’", "ALPHA") .le. f$length("’’a’")
$   then
$     sys_type = "Alpha"
$     alpha    = "YES"
$     def_c_sw = cc_alpha + "/include=dmq$user:"
$   else
$     sys_type = "VAX"
$     alpha    = "NO"
$     def_c_sw = cc_vax + "/include=dmq$user:"
$ endif
$
$ ASK_C_SW:
$ if "''p1'" .eqs. ""
$   then
$     inquire p1 "Enter C compile switches [D:''def_c_sw']"
$     if p1 .eqs. "" then p1 = def_c_sw
$ endif
$ c_sw = "''p1'"
$
$ wl ""
$ wl "------- Build Parameters -------"
$ wl "   CC switches: ''c_sw'"
$ wl "   System type: ''sys_type'"
$ wl ""
$
$ call CC x_attnam
$ call CC x_attnum
$ call CC x_atttmp
$ call CC x_basic
$ call CC x_exit
$ call CC x_get
$ call CC x_getall
$ call CC x_getem
$ call CC x_getpri
$ call CC x_getsel
$ call CC x_getsho
$ call CC x_getw
$ call CC x_locate
$!** call CC x_putdlj
$ call CC x_putslf
$ call CC x_recovr
$ call CC x_select
$ call CC x_shopnd
$ call CC x_timer
$ DONE:
$ wl ""
$ wl "Finished building MessageQ standard examples"
$ exit
$
$!================================================================
=
$
$ ! p1 = program to compile
$
```

```
$ CC:  subroutine
$ on warning then exit 4
$ on control_y then exit ’ss$_abort’
$
$ wl "Building ’’P1’..."
$
$ if f$search("’’p1’.obj") .nes. "" then delete/nolog ’p1’.obj.*
$ if f$search("’’p1’.lis") .nes. "" then delete/nolog ’p1’.lis.*
$ if f$search("’’p1’.exe") .nes. "" then delete/nolog ’p1’.exe.*
$ if f$search("’’p1’.map") .nes. "" then delete/nolog ’p1’.map.*
$ if f$search("’’p1’.dia") .nes. "" then delete/nolog ’p1’.dia.*
$
$ cc’c_sw’ ’p1’
$
$ if f$search("’’p1’.obj") .nes. ""
$   then
$     link ’p1’,dmq$lib:dmq/opt
$ exit
$ endsubroutine
```

BEA MessageQ for OpenVMS allows two kinds of application linking: linking with the run-time library (RTL) and linking with the object library. The BEA MessageQ run-time libraries must be installed before linking applications.

Linking with the Run-Time Library

The run-time library (RTL) is the standard form of linking application modules with the BEA MessageQ environment. The files required for linking with BEA MessageQ are located in two areas: `DMQ$LIB` and `DMQ$USER`. The `DMQ$LIB` area contains the site-independent files and the `DMQ$USER` area contains the site-specific files that you or your BEA MessageQ system manager customize.

To link BEA MessageQ applications, use the `DMQ$LIB:DMQ/OPT` switch in your linker command line. Use the following command to link your application:

```
$ LINK SAMPLE_C, DMQ$LIB:DMQ/OPT
```

The options file contains all the commands needed to connect to the current version of the BEA MessageQ RTL. RTLs are OpenVMS run-time libraries that allow code sharing between numerous simultaneous users of BEA MessageQ. Using RTLs saves memory, disk space, and link time.

Linking with the Object Library

You can link your BEA MessageQ program using the BEA MessageQ object library instead of the RTL. Using this method, each BEA MessageQ program is built with its own copy of the BEA MessageQ procedures. You can also link with the object library and with partial run-time libraries for protected code and BEA MessageQ Script Facility code.

To link with the object library, use the `DMQ$LIB:DMQ$OLB/OPT` switch in your linker command line. Enter the following command to link your application:

```
$ LINK SAMPLE_C, DMQ$LIB:DMQ$OLB/OPT
```

Note that you may also need to include various language-specific run-time libraries or object libraries depending upon how your OpenVMS system manager has installed your layered languages.

Note: Use object library linking when you need an OpenVMS traceback.

Running a BEA MessageQ Application

Before running a program that uses BEA MessageQ, you must set the environment to identify the message queuing bus and the message queuing group environment with which the program will be associated. See the Connecting to the BEA MessageQ Environment topic for information on how to set environment variables.

To run a UNIX program in the background, enter the following command:

```
#  myprog &
```

where:
`myprog` is the name of your program.

Running an OpenVMS Program as a Detached Process

You can run a detached process with or without a DCL context. If you choose to run your process without a DCL context, you can invoke the command procedure `DMQ$EXE:DMQ$COPY_LNM_TABLE` to copy all the necessary logical names into the group or system logical name table. The detached process will then have access to the logical names defined for BEA MessageQ.

If the process is to be run with DCL context, you can invoke the command procedure `DMQ$SET_LNM_TABLE` before running the image. The command procedure `DMQ$DETACH_PROCESS` in `DMQ$EXE` is an example of invoking `DMQ$SET_LNM_TABLE` and running a detached process. Listing 6-5 shows a sample command procedure fragment that runs `LOGINOUT.EXE` and uses the command procedure `DMQ$DETACH_PROCESS` to run the detached process.

Listing 6-5 Command Procedure to Run as a Detached Process

```
.
.
.
   $ write sys$output "...Starting MY_IMAGE.EXE"
   $ startup_file = "START_TEMP_" + f$getjpi("","PID") + ".COM"
   $ create/owner='f$user()' 'startup_file'
   $ open/append startup 'startup_file'
   $ write startup "$ @DMQ$EXE:DMQ$DETACH_PROCESS 1 1 MY_IMAGE.EXE
DMQ$EXE"
   $ close startup
   $ run sys$system:loginout.exe -
               /input                 = 'startup_file' -
               /output                = MY_PROCESS.LOG -
              /error                 = 'f$trnlnm(""sys$error"")' -
               /process_name          = MY_PROCESS -
               /priority              = 4 -
               /uic                   = 'f$user()' -
               /io_buffered           =   100 -
               /io_direct             =   100 -
               /buffer_limit          =   200 -
               /working_set           =   500 -
               /maximum_working_set   =   700 -
               /extent                =  2000 -
               /page_file             = 10000 -
               /ast_limit             =   100 -
               /buffer_limit          =   100 -
               /enqueue_limit         =   100 -
               /file_limit            =    50 -
               /queue_limit           =   100
.
.
.
```

Running Existing BEA MessageQ Applications Under Version 5.0

To run existing applications under BEA MessageQ Version 5.0, you must begin by converting your group initialization files to the Version 5.0 format and restarting your message queuing groups. Table 6-5 describes whether or not existing applications need to be recompiled or relinked to run under BEA MessageQ Version 5.0:

Table 6-5 Existing Application Recompiling and Relinking Requirements

Product	Running Existing Applications
BEA MessageQ for UNIX	To take advantage of BEA MessageQ Version 5.0 features, you must recompile and relink your applications.
BEA MessageQ for OpenVMS	Applications that are linked with the BEA MessageQ run-time library (`DMQ.OPT`) do not have to relink to use BEA MessageQ Version 5.0. However, if the application was linked with the object libraries (`DMQ$OLB.OPT`), then a relink is required. If the application uses the `LOCATE_Q_REP/RESP` message, these are now RISC-aligned, and you should recompile and relink your application to take advantage of the change.
BEA MessageQ for Windows NT	BEA MessageQ Version 3.2 or earlier applications do not need to be recompiled or relinked. However, to take advantage of the new BEA MessageQ Version 5.0 features, you must recompile and relink your applications.
BEA MessageQ Windows Client	When upgrading to BEA MessageQ Version 5.0 from any previous version of BEA MessageQ, we recommend that you recompile and relink your application to take advantage of new features.

Running Applications Under Windows 95 or NT Systems

Applications built on Windows Version 3.1 systems (16-bit applications) may run on Windows 95 or NT systems. However, there may be some restrictions, for example, PATHWORKS for Windows NT does not support 16-bit applications. We recommend that you recompile and relink your applications under Windows 95 or Windows NT systems.

To convert a 16-bit application to a 32-bit application on Windows 95 or NT systems, you must recompile and relink your application with the 32-bit import library, `DMQCL32.LIB`.

Linking an Application from a BEA MessageQ Client System to a BEA MessageQ Server System

The following information describes how to link applications between BEA MessageQ client and server systems on Windows, UNIX, and OpenVMS platforms.

Windows Systems

On Windows systems, you have a choice of using static or dynamic linking. Applications that use static linking need to be linked with a specific import library to resolve external function calls. Client applications use the import library `DMQCL32.LIB` (assuming the application is 32-bit), and server applications use the import library `DMQ.LIB`.

Another linking method is dynamic run-time linking. Load either the file dmq.dll or dmqcl32.dll at runtime. You need to structure your application to decide which DLL to use. You can do this by setting an ini file, a Registry entry, or a command line argument. With dynamic run-time linking, you do not need to rebuild your application when changing from client systems to server systems, or vice versa.

UNIX Systems

On UNIX systems, applications must use static linking with specific libraries. Client applications must use `libdmqcl.a` or `libdmqcldnet.a`. Server applications must use `libdmq.a`. You need to rebuild your application and link with the file `libdmq.a`, instead of `libdmqcl.a` or `libdmqcldnet.a`.

OpenVMS Systems

On OpenVMS systems, you typically build your application against the run-time library (RTL). Both client and server applications use the logical name `DMQ$ENTRYRTL` to identify which RTL is used. You only need to execute `DMQ$EXE:DMQ$SET_LNM_TABLE<`*`bus`*`><`*`group`*`>` to select the server RTL. No rebuild is required.

Applications can also statically link against the server or client OLB. You will need to relink your application when changing from client to server.

Testing Return Status

Operating systems have different rules concerning what return status indicates a failure and what indicates a success. Under the OpenVMS system, a returned value indicates an error if the low bit is clear (an even number), and a success if the bit is set (an odd number). UNIX based systems typically classify values below zero as failure. Furthermore, systems can use different status values for the same status condition.

Portable programs must use a set of error-checking rules for all environments. For BEA MessageQ software, the following rules exist for all supported systems:

- A return status equal to `PAMS__SUCCESS` indicates unconditional success.
- All success codes have the low bit set (making the values all odd numbers), including `PAMS__SUCCESS`. A success status other than `PAMS__SUCCESS` indicates successful completion, but that additional information exists. The information is represented by the specific status value returned.
- All error codes have the low bit clear (making the values all even numbers).
- A return status of zero is invalid.
- All `PAMS__*` symbol definitions exist on all BEA MessageQ systems. However, they do not always contain the same numeric values, and all defined symbols will not be returned on all platforms.

Note: Portable code should not use the OpenVMS specific symbol `SS$_NORMAL` when referring to BEA MessageQ functions. Instead, use the BEA MessageQ symbol `PAMS__SUCCESS`.

Listing 6-6 shows how to test for a return status on any BEA MessageQ platform.

Listing 6-6 Portable Code for Testing Return Status

```
EXAMPLE 1 - Simple test for success or failure

status = pams_put_msg(msg_area, pri, target, class, type, del,
    msize, 0,0,0,0);

if ((status & 1) == 0)                       /* Successful? */
{
    printf("%Unexpected error %d returned\n", status);
    exit(1);
}
```

```
EXAMPLE 2 - Testing for various conditions

status = pams_put_msg(msg_area, pri, target, class, type, del,
msize, 0,0,0,0);

if (status != PAMS__SUCCESS)
{
    if (!(success & 1))
    {
        printf("%Unexpected error %d returned\n", status);
        exit(1);
    }

    if (status == PAMS__JOURNAL_ON) printf("Journaling enabled\n");
    /* Successful, and notification that journaling was enabled */
}
/* Continue processing */
```

```
EXAMPLE 3 - Using case statements

status = pams_put_msg(msg_area, pri, target, class, type, del,
    msize, 0,0,0,0);

switch (status)
{

    case PAMS__SUCCESS :
        break;

    case PAMS__JOURNAL_ON :
        printf("Journaling enabled\n");
        break;

    case PAMS__PAMS_DOWN :
        printf("Message bus is down\n");
        exit(1);

    case else :
        printf("PAMS call returned unknown error %d, aborting!\n",
            status);
        exit(1);
}
```

```
/* Continue processing */
```

BEA MessageQ allows OpenVMS systems to automatically translate return status codes in textual messages when an application program exits. To enable this feature, enter the following command before running an application program:

```
$ SET MESSAGE DMQ$MSGSHR
```

If your application aborts during testing, it is useful to have OpenVMS traceback information. The use of the BEA MessageQ object library and the inclusion of symbols in the executable (via the `/DEBUG` switch in the compile step) add to the information that is returned during traceback. If you link your application with the BEA MessageQ RTLs, the traceback line number information is lost. To get a complete traceback, the image must be linked using the object library described in Linking with the Object Library.

Using the BEA MessageQ Test Utility

Using a graphical or character-cell interface, the BEA MessageQ Test utility allows developers to send and receive messages between applications to:

- Build interactive tests of application modules.
- Simulate send and receive messages to any target from any source.
- Exercise the queues in the BEA MessageQ system.

The BEA MessageQ Test utility enables application developers to interactively attach to a permanent or temporary queue, read messages from a script file or available interprocess messages, and pass messages to a defined target queue. Messages sent or received using the Test utility can be previewed using its message display or the echo feature of the Script Facility. The Test utility is available on UNIX, Windows NT, and OpenVMS systems.

To invoke the Test utility using the Motif user interface on UNIX systems, set the environment variables for the bus and group ID and enter the command:

```
dmqtestm
```

To invoke the Test utility on Windows NT systems, enter the following commands:

```
set DMQ_BUS_ID bus_id
set DMQ_GROUP_ID group_id
dmqtestw
```

To invoke the character-cell user interface on UNIX systems, set the environment variables for the bus and group ID and then enter the following command:

```
dmqtestc
```

To access the character-cell Test utility on OpenVMS systems, choose the Test option from the BEA MessageQ main menu. The system prompts you for the following information:

- Queue number—Enter the number of the sender's queue.
- Queue name—Enter the name of the sender's queue.
- Queue type PQ [`Y/N`]—Enter `Y` for primary queue or `N` if you want a secondary or multireader queue.
- Name scope LOCAL [`Y/N`]—Enter `Y` if the queue is a local group name or `N` if it is a remote group name.
- Target group number—Enter the target group number.
- Target process number—Enter the target process number.

Enter a setting at each system prompt or press Return to accept the default settings. Table 6-6 shows the default settings for using the Test utility.

Table 6-6 Test Utility Default Settings

Setting	Default Value
`send_class`	1
`send_type`	-100
`send_priority`	0
`rcv_priority`	0 (all priorities)
`rcv_timeout`	5 seconds
`delivery`	`PDEL_MODE_NN_MEM`

Debugging BEA MessageQ Applications

BEA MessageQ offers a feature called tracing to log internal messaging events to a file as they happen. You can use this file to diagnose application failures as you debug your application. It is important to consider that message tracing generates a high volume of output; therefore, you should only enable tracing for diagnostic purposes in the event of a problem.

BEA MessageQ provides an execution tracing facility for diagnostic purposes. Tracing produces a time-stamped output file showing the sequence of BEA MessageQ function calls and return status codes. If the `DMQ_TRACE_PREFIX` environment variable is set, tracing information goes to `$DMQ_TRACE_PREFIX.pid`. If it is not set and the `DMQ_TRACE_FILE` environment variable is set, then tracing information will go to `$DMQ_TRACE_PREFIX`. Otherwise, tracing information will go to the standard output (this is not desirable on Windows NT). Each message will contain a time stamp if the `DMQ_TRACE_TIMESTAMPS` environment variable is set.

Tracing Messages on UNIX Systems

Some special features are incorporated into BEA MessageQ to aid in debugging. The `PAMS_TRACE` environment variable allows you to enable tracing to BEA MessageQ callable services. To enable `PAMS_TRACE` on your system, enter the following command:

```
# setenv PAMS_TRACE 1
```

BEA MessageQ logs trace information to the standard output unless `DMQ_TRACE_FILE` is set. Following is the trace information for a `pams_put_msg` call with a 30-second timeout from source 1.1 to target queue 1:

```
PAMS:PAMS-Timeout was ZERO, using 30 seconds
PAMS:PAMS-****** sending message ******
PAMS:PAMS-Source      :, 65537 (10001)
PAMS:PAMS-Target      :, 1 (1)
PAMS:PAMS-Type / Class:, 6488162 (630062)
PAMS:PAMS-Delivery    :, 39 (27)
PAMS:PAMS-UMA         :, 5 (5)
PAMS:PAMS-Resp Q      :
```

```
PAMS:PAMS-******************************
PAMS:PAMS-PAMS_put_cleanup
```

To disable `PAMS_TRACE`, enter the following command:

```
# unsetenv PAMS_TRACE
```

Tracing Messages on Windows NT Systems

Tracing is enabled by setting environment variables using the following command:

```
SET PAMS_TRACE=value
```

where *`value`* is an arbitrary value.

To disable a trace, set the variables to a null value, as follows:

```
SET PAMS_TRACE=
```

You can check your environment variables at any time by entering `SET` at the command line.

Tracing Messages on OpenVMS Systems

On OpenVMS systems, you can activate tracing using the `DMQ$DEBUG` logical name. Once tracing is enabled, you can direct trace output using the `DMQ$TRACE_OUTPUT` logical name. For more detailed information about how to troubleshoot BEA MessageQ errors on OpenVMS systems, see the *BEA MessageQ Configuration Guide for OpenVMS.*

Controlling Message Flow

When the message queuing environment becomes congested, BEA MessageQ lets you control the flow of messages by setting environment variables that restrict messaging rates on a per-process basis.

Note: This feature is available only on BEA MessageQ for UNIX.

BEA MessageQ uses a congestion control algorithm to reduce the number of messages being enqueued, which allows the system to process the backlog of messages.When the congestion condition subsides, BEA MessageQ gradually raises the rate of message flow back to the maximum flow rate set for the queue.

The rules for enforcing congestion control are as follows:

- While a congestion condition exists, the enqueue and dequeue rate of all processes is monitored.
- The maximum rate at which any process can enqueue messages during a congestion condition is equal to a maximum value that is adjusted at regular intervals.
- If, during a period of congestion, a process enqueues more messages than it dequeues, its message flow rate is reduced by a percentage of its current flow during the next interval.
- After a period of congestion has subsided, the flow rate of each process is adjusted upward until the flow rate exceeds the maximum congestion flow rate (`DMQ_FLOW_MAXIMUM`). At this point, flow control is no longer enforced.

The following table lists the congestion control environment variables for BEA MessageQ for UNIX systems.

Environment Variable	Description
`DMQ_FLOW_MAXIMUM`	Maximum number of messages per second a process can enqueue during a period of congestion. The default value is set to 1000 messages per second.
`DMQ_FLOW_MINIMUM`	Minimum number of messages per second a process can enqueue during a period of congestion. BEA MessageQ will always allow you to enqueue at least this number of messages per second. The default value is set to 10 messages per second.
`DMQ_FLOW_INTERVAL`	Interval at which BEA MessageQ checks message flow and makes adjustments to the current flow rate. The value is expressed in milliseconds. The default value is set to 250 milliseconds.
`DMQ_FLOW_INCREASE`	Number of messages per second to increase the flow rate at the current interval after a congestion period has subsided. The default value is set to 10 messages.

Environment Variable	Description
`DMQ_FLOW_DECREASE`	Percent reduction of the current flow rate to apply at each interval during periods of congestion. The value must be specified as a real number in the range (0.0 to 1.0). If the value is set to zero, then the maximum flow rate for the given process is equal to the flow maximum as defined by the environment variable `DMQ_FLOW_MAXIMUM`, and is not adjusted downward at each flow interval. The default value is set to 0.25 intervals.

To specify congestion control for an application, use the following syntax to set the appropriate environment variable prior to starting the application:

C shell

```
setenv DMQ_FLOW_MAXIMUM 500
```

Bourne shell

```
DMQ_FLOW_MAXIMUM=500
export DMQ_FLOW_MAXIMUM
```

Because the environment variables are set on a per-process basis, you can set different values for each application in the environment.

7 Using the Script Facility

The BEA MessageQ Script Facility provides a powerful tool for application developers to use in simulating message exchange between programs. Instead of writing a test program, you create a **script file** containing instructions for capturing messages sent or received by an application, replaying captured messages, or simulating messages sent from an application that is still under development.

Application developers can use the Script Facility to:

- Simulate messages sent to an application without writing a test program
- Selectively trace messages sent or received by an application and display them on the screen or log them to a file
- Capture message traffic and replay the log files to support concurrent development and testing of applications
- Simulate message traffic between client/server application components still under development
- Create message trace files that assist developers in debugging applications based on message traffic
- Stress test applications under different load levels by generating high levels of message traffic
- Facilitate the development of large-scale integrated applications by simulating message traffic from remote components

Instructions are entered to the script file using the BEA MessageQ scripting language. When script processing is enabled, BEA MessageQ processes the script file and executes the instructions.

If you need to view or record the exchange of messages between applications under development, you can use the Script Facility to capture messages sent or received by an application. Captured messages can be displayed on a monitor or written to a log file. Message capture documents messages sent and received by an application, enabling developers to debug message exchange.

The BEA MessageQ Script Facility message replay feature is like using a tape recorder with messaging. First, using message capture, you record the messages sent or received by an application. Then, using replay, you send the messages captured in the log file as input to another application. Message replay can be used to debug message exchange between applications that are still under development.

Scripts can also be used to create a message to be sent. For example, if a sender program is under development, you can create a script file to simulate the messages that it will send. Then, when you enable script processing, the messages contained in the script file are delivered to the receiver program to test its response.

Note: The BEA MessageQ Script Facility is available on UNIX and OpenVMS systems only.

How to Use the Script Facility

Use the scripting language commands to create script files that send messages, capture messages, or both. You can add instructions to the script file to repeat an operation, add a time delay between functions, or add comments to document the script file.

After you create the script file, you can use the Script Facility to verify that the syntax of the file is correct. If errors exist in the scripting language commands, BEA MessageQ will highlight the line numbers and describe the errors to help you debug your BEA MessageQ script.

When your script file is correct and ready for use, you enable script processing by setting the Script Facility environment variable to the name of the script file or the log file of captured messages to be used as input. When you run your application with the environment variable set, BEA MessageQ reads the script file, delivers the defined messages to the target queues, and captures messages as specified.

Using the BEA MessageQ Scripting Language

BEA MessageQ script files are ASCII files created using a text editor. Though the content must adhere to the scripting language syntax, it is not case sensitive and does not require that data be entered in specific column positions in the file. When including a group name in a script, the group name must start with a letter. Group names beginning with a number or special character are not allowed.

On OpenVMS systems, you create a script file using a .PSS file extension. On UNIX systems, you create a script file using a .pss file extension. Use tabs and spaces within the script file to make it easier to read. See the Adding Repeats, Delays, and Comments to Scripts topic for more information on how to add comments to a script file to annotate its purpose and use.

The BEA MessageQ scripting language uses commands to identify the functions to be performed. Table 7-1 describes information on BEA MessageQ Script Facility commands:

Table 7-1 BEA MessageQ Script Facility Commands

Function	Command Begin/End	Modifiers	Description
Send a message	`MSG/EOM`		Identifies the beginning and end of the message header and content.
Capture messages sent	`SET SEND`		Sets message capture to include messages sent by the application.
		`ECHO`	Displays messages on the screen.
		`ECHO LINES=`*n*	Selects the number of lines displayed.
		`LOG`	Writes messages to a log file.
		`LOG LINES=`*n*	Selects the number of lines logged.
		`OFF`	Captures messages sent by the application in a log file only; does not send messages to the target queue.

Table 7-1 BEA MessageQ Script Facility Commands

Function	Command Begin/End	Modifiers	Description
		`ON`	Sends messages to the target queue and captures them in a log file. `SET SEND ON` is the default action for this command.
Capture messages received	`SET RECEIVE`		Sets message capture to include messages received by the application.
		`ECHO`	Displays messages on the screen.
		`ECHO LINES=`*`n`*	Selects the number of lines displayed.
		`LOG`	Writes messages to a log file.
		`LOG LINES=`*`n`*	Selects the number of lines logged.
		`OFF`	Prevents the application from receiving messages from sources other than the script file.
		`ON`	Application receives messages from all processes. `SET RECEIVE ON` is the default action for this command.
Set the log file name	`SET LOG`	*`file_name`*	Specifies the log file name. `SET LOG` must precede the `SET SEND` or `SET RECEIVE` commands in the script.
Add messages to an existing file	`SET LOG`	*`file_name`* `APPEND`	Adds messages to an existing log file.
Add comments	`COMMENT/ENDC`		Designates the beginning and end of comments to explain what the script file does.
Set a time delay	`DELAY`	`time`	Creates a time delay, which is useful to simulate message arrival patterns.

Table 7-1 BEA MessageQ Script Facility Commands

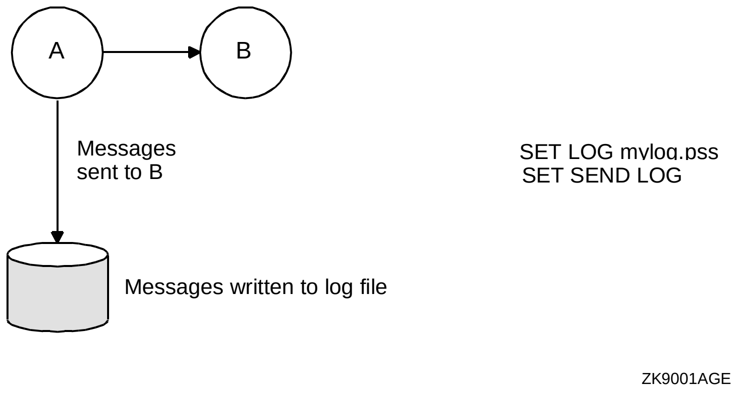
Function	Command Begin/End	Modifiers	Description
Repeat an operation	`REPEAT/ENDR`		Creates a repeat loop construct.

Capturing, Replaying, and Simulating Message Exchange

The BEA MessageQ Script Facility is most commonly used to capture messages sent or received by an application. Captured messages document message exchange and can be used as the input stream to another application to test its response.

For example, if you are testing message exchange between two running applications, you can use a script file to capture the output of the sender program. Figure 7-1 shows application A sending messages to application B and recording those messages in a log file.

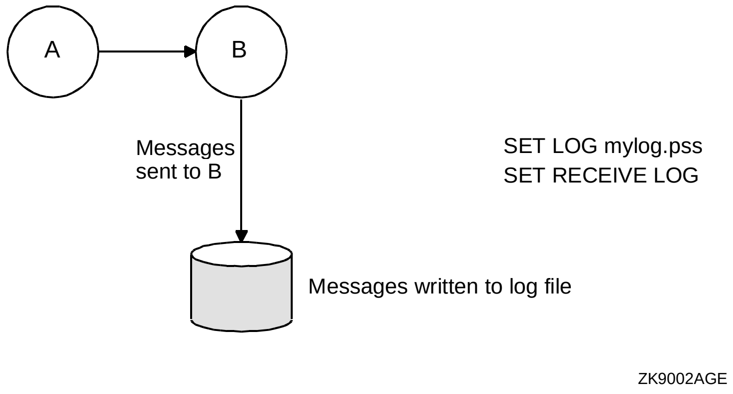
Figure 7-1 Sending Messages and Capturing Output



The log file of captured messages can be used to document the messages sent by A. It can also be used as an input stream to B during testing if application A is not always available to send messages.

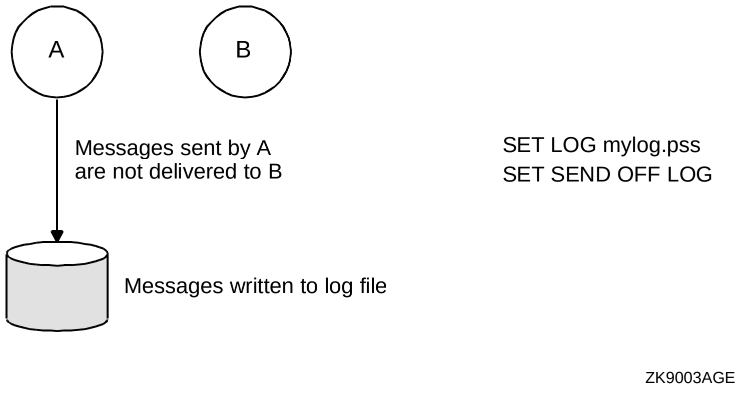
Depending on the requirements of your test environment and applications, you can choose to capture messages received rather than capturing messages sent. Figure 7-2 shows how to use a log file to capture messages received by an application.

Figure 7-2 Sending Messages and Capturing Input



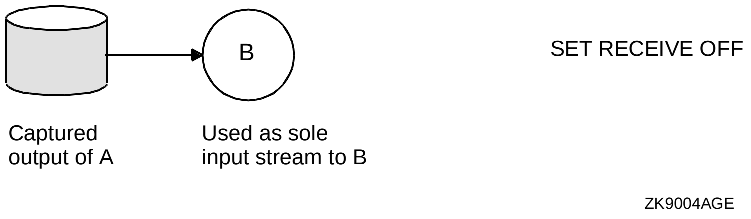
In this example, the log of messages received by application B matches the log file of messages sent by application A. You can also use the Script Facility when one of your applications is not running. For example, Figure 7-3 shows how application A can capture messages it sends in a log file without BEA MessageQ delivering the messages to application B.

Figure 7-3 Capturing Output Without Sending Messages


A
B
Messages sent by A are not delivered to B
SET LOG mylog.pss
SET SEND OFF LOG
Messages written to log file
ZK9003AGE

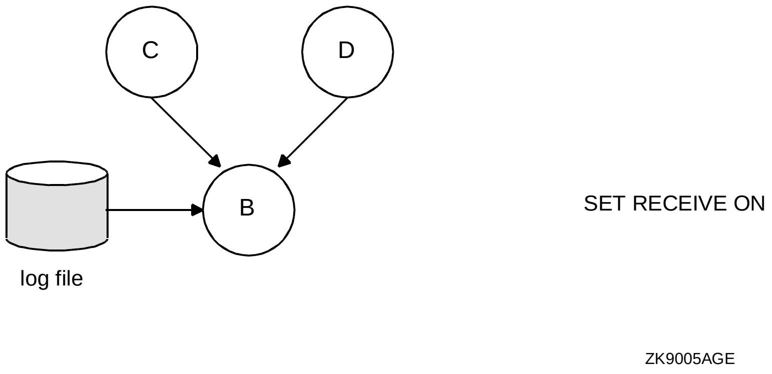
Then when application B is ready to test, you can use the script file containing messages sent by application A to test it. Figure 7-4 shows how to replay messages and to restrict application B to receiving only messages from the script file.

Figure 7-4 Replaying Captured Messages



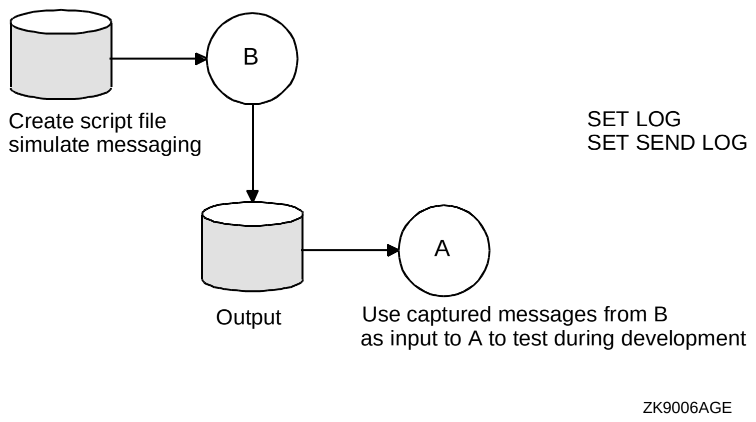
Or, you can have the receiver program obtain messages from the script and messages from other applications as shown in Figure 7-5.

Figure 7-5 Receiving Messages from Applications and Scripts



And, if the receiver program is ready for testing, but the sender program is not, you can create a script file to simulate message exchange. If you capture the output of application B during this process, you can use it as input to application A when it is ready for testing as shown in Figure 7-6.

Figure 7-6 Writing Scripts to Send and Capture Messages



For some programs, script output is buffered, depending on the operating system and whether the program is running in the background. For example, on a Solaris system, output from `dmqcls` is unbuffered, but output from `dmqtest` is buffered.

The remaining sections of this topic provide more detailed information and examples of how to create script files.

Capturing Messages Using Scripts

The `SET` command is used to select messages for capture. The `SET SEND` command captures output by recording the messages sent by an application. The `SET RECEIVE` command captures input by recording the messages received by an application. The `SET` command uses the following syntax:

```
SET RECEIVE  modifier [FROM MessageQ address]
SET SEND     modifier [TO MessageQ address]
```

The modifiers to these commands are as follows:

- `ON/OFF`—determines whether messages are sent to the target or only to the log file, and whether the receiver program receives all input or input only from the script file
- `ECHO`—displays captured messages on the screen

- `LOG`—writes captured messages to the specified log file

The `FROM/TO` address qualifier is the queue address of the message to which messages will be sent or from which messages will be read when the script is run.

Controlling Message Delivery Using Scripts

Using the `ON/OFF` modifiers with the `SET SEND` and `SET RECEIVE` commands, you can control the delivery of messages from the script file and from other sources. Following is a list of valid commands that you can enter to your script file to control message delivery with scripts:

Command	Description
`SET SEND OFF`	Captures messages sent by the application but does not deliver them to the target queue.
`SET SEND ON`	Captures messages sent by the application and delivers them to the target queue. `ON` is the default for the `SET SEND` command.
`SET RECEIVE OFF`	Captures messages received by the application, but restricts the application to receiving only those messages sent from the script.
`SET RECEIVE ON`	Captures messages received by the application from the script and any other source. `ON` is the default for the `SET RECEIVE` command.

Displaying Captured Messages on the Screen

To display captured messages on the screen, use the `ECHO` modifier with the `SET SEND` or `SET RECEIVE` commands. Use the `ECHO LINES=`*`n`* modifier if you only want to display a specified number of lines of the message. Following is a list of valid commands that you can enter to your script file to display messages on the screen:

Command	Description
`SET SEND ECHO`	Displays the messages sent by the application to the screen.
`SET SEND ECHO LINES=`*`n`*	Displays *n* lines of the messages sent by the application to the screen.
`SET RECEIVE ECHO`	Displays the messages received by the application to the screen.
`SET RECEIVE ECHO LINES=`*`n`*	Displays *n* lines of the messages received by the application to the screen.

Writing Captured Messages to a Log File

To write messages to a log file, begin by specifying the name of the log file. To create a new log file to store captured messages, use the following command:

```
SET LOG file_name
```

Note: File names are case sensitive on UNIX systems. Enter the file name with the exact upper- and lowercase letters that you will use to retrieve the file. The file name can be specified with a path name or directory name to store it in a specific area. Both absolute and relative path names can be used.

Note: If `SET LOG` is used in a script, the `pams_get_msg` call used to activate scripting must be at least 1036 bytes in size.

To add captured messages to an existing log file, use the following command:

```
SET LOG file_name APPEND
```

If you do not provide a file extension, `.LOG` is used by default on OpenVMS systems and .log is used by default on UNIX systems. If you want to replay the captured messages, use `.PSS` (OpenVMS) or `.pss` (UNIX) as the file extension to distinguish the log file as an input file.

The beginning of each log file has a comment line containing the date and time it was created. A comment line is added each time the file is reopened. On UNIX systems, only one log file can be open at a time. On OpenVMS systems, a maximum of four log files can be open at a time.

To write captured messages to a log file, use the `LOG` modifier with the `SET SEND` or `SET RECEIVE` commands. Use the `LOG LINES=`*`n`* modifier if you want to log only a specified number of lines of the message. Following is a list of valid commands that you can enter to your script file to log messages to a file:

Command	**Description**
`SET SEND LOG`	Writes the messages sent by the application to the specified log file.
`SET SEND LOG LINES=`*`n`*	Logs *n* lines of the messages sent by the application to the specified log file.
`SET RECEIVE LOG`	Logs the messages received by the application to the specified log file.
`SET RECEIVE LOG LINES=`*`n`*	Logs *n* lines of the messages received by the application to the specified log file.

Listing 7-1 shows the syntax of a BEA MessageQ script file that creates a log file named `MYLOG.PSS`, captures messages sent and received by the application, and displays them on the screen.

Listing 7-1 Sample Script to Capture Messages

```
COMMENT
    Example MessageQ script source file to capture messages,
    display them on the screen, and log them to a file.
ENDC
SET LOG MYLOG.PSS
SET RECEIVE ECHO LOG
SET SEND ECHO LOG

 MSG
      TARGET=MHIS_EK_INTERFACE     SOURCE=MHIS_REQ_PROCESSOR
      CLASS=PAMS                   TYPE=ASRS_PERF_DATA_REQ
      A '1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 EOM
```

Listing 7-2 shows the content of MYLOG.PSS created when script processing is enabled using the script file in the previous example.

Listing 7-2 Sample Log Generated by a Script File

```
!*** Session begun at 22-MAR-1994 10:37:23.95 *******************

MSG             ! Message receive at 22-MAR-1994 10:37:26.21
    SOURCE = 20,1                TARGET = 30,1
    CLASS = PAMS                 TYPE = ASRS_PERF_DATA_REQ
    XB 31, 32, 33, 34, 35, 36, 37, 38, 39, 30       !’1234567890’
    XB 20, 41, 42, 43, 44, 45, 46, 47, 48, 49       !’ ABCDEFGHI’
    XB 4A, 4B, 4C, 4D, 4E, 4F, 50, 51, 52, 53       !’JKLMNOPQRS’
    XB 54, 55, 56, 57, 58, 59, 5A,                  !’TUVWXYZ’
EOM
```

Though the format of the message data in the log file varies somewhat from a script file, it can be used exactly as a script file to simulate message exchange. Use a log file as input by setting the BEA MessageQ environment variable DMQ_SCRIPT to equal the log file name. Then run the test application and it will receive and process the messages contained in the log file.

Writing Captured Messages to Multiple Log Files

On UNIX systems, only one log file can be open at a time. However, the BEA MessageQ Script Facility on OpenVMS systems lets you log messages to multiple log files simultaneously. Listing 7-3 shows how to write messages received by an application to one log file (RECEIVE.PSS) while writing messages sent by the application to another log file (SEND.PSS).

Listing 7-3 Sample Script Using Multiple Log Files

```
COMMENT
    Example MessageQ script source file WITH LOGGING TO
    MULTIPLE LOG FILES
ENDC

SET LOG RECEIVE.PSS
SET RECEIVE LOG

SET LOG SEND.PSS
SET SEND LOG

REPEAT 5
 MSG TARGET=MHIS_EK_INTERFACE SOURCE=MHIS_REQ_PROCESSOR
      CLASS=PAMS              TYPE=ASRS_PERF_DATA_REQ
        A ’1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ’
        A ’MSG FROM REPEAT NUMBER 1 - WHICH IS SENT 5 TIMES’
        S 1
    EOM
ENDR
```

Replaying Messages

To use the BEA MessageQ Script Facility, you set an environment variable on UNIX systems or a logical name on OpenVMS systems to the name of the script file or the log file that you want to use as input to the application being tested. When you run the application after the environment variable is set, the Script Facility reads the script file or log file and uses the `pams_put_msg` function to deliver the messages contained in the file to the target queue.

Note: The script file may be only one of many sources of messages sent to the application. If messages are delivered to the application’s primary queue from sources other than the script file, these messages will also be read and processed.

If the script file requests messages to be captured, the Script Facility signals the application to use BEA MessageQ logging routines that write the messages sent or received by the application to the designated log file.

Script Processing on UNIX Systems

Script processing on UNIX systems is enabled by defining the environment variable `dmq_script` as the script file name or log file name that you want to use as input to the program being tested. Before setting the `DMQ_SCRIPT` environment variable, you must first set the `BUS` and `GROUP_ID` environment variables. Use the following commands to set the environment variable to enable script file processing. The command using `csh` syntax is:

```
setenv DMQ_SCRIPT mylog.pss
```

The command using `sh` syntax is:

```
DMQ_SCRIPT=mylog.pss
export DMQ_SCRIPT
```

Define the `DMQ_SCRIPT` environment variable after running the Group Control Process (`dmqgcp`) to boot the system. If `DMQ_SCRIPT` is defined before booting the system, processing a script produces error messages for each line of the script.

When you run the application with BEA MessageQ script processing enabled, BEA MessageQ translates this symbol when the `pams_attach_q` function is called. BEA MessageQ processes the script, directing messages to their target queues and turning on message logging, if applicable. Script processing on UNIX systems begins when the target process issues a `pams_get_msg` or `pams_get_msgw` call.

Client programs do not access the `DMQ_SCRIPT` environment variable or perform script processing directly. Instead, the client program uses the associated Client Library Server (CLS) to perform script processing. Writing to the log file or echoing output is performed relative to the CLS rather than the client program.

The Script Facility on UNIX systems also allows developers to initiate script processing for an application that is currently running. To enable script processing for a running application, use the dmqscript utility to direct a script file to the target queue of the application.

To turn on script processing, the script file must begin with the command `SET SCRIPTS ON`. To turn off script processing, the script must contain the `SET SCRIPTS OFF` command. Table 7-2 describes the script control commands which are only available on BEA MessageQ for UNIX systems.

Table 7-2 Script Control Commands (UNIX only)

Function	Command Begin/End	Modifier	Description
Enable script processing	`SET SCRIPTS`	`ON`	This command is sent to an application that is already running, enabling it to receive messages from a script file or begin capturing messages.
Disable script processing		`OFF`	This command turns off script processing for a running application. The application no longer receives messages from the script file and stops capturing messages.

Listing 7-4 provides a sample script that turns on script processing to begin message logging for the running application.

Listing 7-4 Turning On Scripts for a Running Application

```
SET SCRIPTS ON
SET LOG /mypath/mylog.log
SET SEND LOG LINES=999
SET RECEIVE LOG LINES=999
```

To process a script file, use the following command syntax:

`dmqscript -f` *`script_file_name`* `-q` *`nn`*

where:

`-f` *`script_file_name`*	Provides the name of the script file to process. The default extension for script files is `.pss`.
`-q` *`nn`*	Specifies the queue number of the application to which the script control commands `SET SCRIPTS ON` or `SET SCRIPTS OFF` should be directed.

Script Processing on OpenVMS Systems

BEA MessageQ for OpenVMS software enables script file processing when the logical name `DMQ$SCRIPT` is defined as a file name or as the word `YES`. The name of the script file to process can be specified in one of the following ways:

- Define the logical name `DMQ$SCRIPT` to pass the script file name directly to the Script Facility as follows:

  ```
  $ DEFINE DMQ$SCRIPT script_file_name
  ```

- Then run the application that you want to test using the script file as input.
- Set the logical name `DMQ$SCRIPT` to `YES`, run the application that you want to test, and enter the script file name in response to the prompt as follows:

  ```
  $ DEFINE DMQ$SCRIPT YES
  $ RUN application_name
  Script file: script_file_name
  ```

- Define a DCL foreign command to invoke an image file name. The script file name then can be entered directly on the DCL command line, as follows:

  ```
  $ DEFINE DMQ$SCRIPT YES
  $ ifn:==$drive_name:[directory_name]application_name
  $ ifn script_file_name
  ```

When you use the Script Facility on BEA MessageQ for OpenVMS systems, all messages defined in the script file are delivered to the target queue of the application program you run regardless of the specified message `TARGET` argument specified in the message header phrase.

To stop script file processing, use the `DEASSIGN` command as follows:

```
$ DEASSIGN DMQ$SCRIPT
```

The `DMQ$EXAMPLES` directory contains a program called `sender.c` that enables application developers to set the target queue used with script processing. In addition, this program enables an application to read messages from a script file and forward them to a program that is already running.

Writing Scripts to Send Messages

If you are unable to create a script file using message capture, you can use the BEA MessageQ scripting language to create a new file defining the message that you want to send. When script processing is enabled, BEA MessageQ sends the message to a target queue where it is read by the application being tested.

To create a script file that sends a message to a target queue, use the scripting language to:

1. Designate the beginning and end of the message
2. Specify the source, target, type and class descriptors that form the message header
3. Create the message content

Defining Messages in Scripts

To define a message, enter the following to the script file:

- The `MSG` command to designate the beginning of the message definition
- The message header information including the target, source, class, and type of the message
- The message data
- The `EOM` command to designate the end of the message definition

Listing 7-5 illustrates a BEA MessageQ script file for sending a message to a target queue. The message in this example sends the numbers “0–9” and the letters “A–Z” to a target queue number 1 in group 30.

Listing 7-5 Sample Script to Send a Message

```
COMMENT
    Example MessageQ script source file to send a message
ENDC

 MSG
      TARGET=30,1
      SOURCE=20,1
      CLASS=PAMS
      TYPE=ASRS_PERF_DATA_REQ
      A '1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 EOM
```

Defining the Message Header

To form a message header, the BEA MessageQ scripting language uses descriptors to designate the target, source, class, and type arguments for the `pams_put_msg` function. Note that the equal sign (=) is optional, and the commands `FROM` and `TO` can replace the commands `SOURCE` and `TARGET`. Listing 7-6 shows the format of the message header.

Listing 7-6 Message Header Format

```
MSG

   TARGET  = {MessageQ address}
   SOURCE  = {MessageQ address}
   CLASS   = {PAMS class number}
   TYPE    = {PAMS type number}

EOM
```

The message header descriptors require the following input to specify the `pams_put_msg` arguments:

Argument	Description
`TARGET`	The queue address to which the messages in the script file are sent. The Script Facility allows the `PAMS_` prefix to be omitted. On OpenVMS systems, the script file messages are directed to the primary queue of the running application regardless of the target queue specified.
`SOURCE`	Queue address of the message source. The script facility allows the `PAMS_` prefix to be omitted.
`TYPE`	Descriptor identifying the message type. The Script Facility allows the `MSG_TYP_` prefix to be omitted.
`CLASS`	Descriptor identifying the message class. The script facility allows the `MSG_CLS_` prefix to be omitted.

Additional Arguments for UNIX Systems

In addition to the target, source, class, and type descriptors in the message header, the Script Facility on BEA MessageQ for UNIX systems offers descriptors to specify the delivery, undeliverable message action (UMA), and priority arguments for the `pams_put_msg` function. Valid values for the delivery mode and UMA can be found in the `p_symbol.h` include file.

These additional UNIX message header descriptors require the following input to specify the `pams_put_msg` arguments:

Argument	Description
`DELIVERY`	Value for the delivery mode as defined in the `p_symbol.h` include file.
`UMA`	Value for the undeliverable message action as defined in the `p_symbol.h` include file.
`PRIORITY`	Message priority, where 0 is the lowest priority and 99 is the highest priority.

Listing 7-7 shows the format of a complete UNIX message header.

Listing 7-7 UNIX Message Header Format

```
MSG

   SOURCE  = {MessageQ address}
   TARGET  = {MessageQ address}
   CLASS   = {PAMS class name}
   TYPE    = {PAMS type name}
   DELIVERY = {MessageQ delivery mode value}
   UMA  = {MessageQ undeliverable message action value}
   PRIORITY = {MessageQ priority}

EOM
```

Defining the Message Data

This topic describes the valid syntax for specifying message content. The BEA MessageQ scripting language syntax requires you to specify the data format, data type, and content of the message.

The valid data formats are:

- `D`—Decimal
- `X`—Hexadecimal
- `O`—Octal
- `Z`—Zero-fill
- `A`—ASCII
- `S`—ASCII space-fill

The binary data formats allow the specification of bytes, words, and longwords. Data types for each data format are described in the script file as follows:

- `B`—A list of 8-bit bytes
- `W`—A list of 16-bit words
- `L`—A list of 32-bit longwords

The content of the message is listed after the data format and data type codes. A comma (`,`) must separate values in the value list. Each value cannot exceed the maximum unsigned value that may be stored in the selected data field.

Table 7-3 lists the valid syntax and provides examples for how to specify message content.

Table 7-3 Valid Message Data Syntax

Data Format	Syntax/Description
Decimal Binary Data	`D` (`B/W/L`) <*`SIGNED_NUMBER`*>, ... <*`SIGNED_NUMBER`*> The values are stored in the message in binary format. The word *decimal* applies only to the base used in entering the data values in the script file. The values are not stored in packed decimal format.
Hexadecimal Binary Data	`X` (`B/W/L`) <*`HEX_NUMBER`*>, ... <*`HEX_NUMBER`*> The values are stored in the message as unsigned hexadecimal values.
Octal Binary Data	`O` (`B/W/L`) <*`OCTAL_NUMBER`*>, ... <*`OCTAL_NUMBER`*> The numeric values in the octal binary data phrase are unsigned octal numbers.
Zero-Fill Binary Data	`Z` (`B/W/L`) <*`NUMBER`*>, ... <*`NUMBER`*> The values are stored in the message as unsigned decimal values.
ASCII Data	`A`(<*`NUMBER`*>)'<*`ASCII_CHARACTERS`*>' `A`(<*`NUMBER`*>)"<*`ASCII_CHARACTERS`*>" An unsigned decimal value specifying the number of blanks to fill into successive fields of the size specified by data type. This format allows the text string to be left-justified into a field <*`NUMBER`*> length long. This allows easy space-filling of a field after the text string. Characters in the quoted string fill into successive bytes starting at the current position in the message text. Note that spaces and tabs are significant when enclosed in quoted strings and that the case of characters in quoted strings is preserved.

Table 7-3 Valid Message Data Syntax

Data Format	Syntax/Description
ASCII Space-Fill Data	`S<`*`NUMBER`*`>` An unsigned decimal value that specifies the number of spaces to fill into successive bytes starting at the current position in the message text

Adding Repeats, Delays, and Comments to Scripts

In addition to commands for sending and capturing messages, you can add instructions to scripts that enable them to better simulate production conditions during testing. This topic describes how to:

- Repeat an operation in a script
- Enter time delays to simulate message arrival patterns
- Add comments to document script functions

Repeating an Operation

The `REPEAT` and `ENDR` commands begin and end repeat groups. A repeat group allows messages to be repeated. The format for using this command is as follows:

```
REPEAT <n>
     MSG
       .
       .
       .
     EOM
ENDR
```

The message contained between the `REPEAT` *n* and the `ENDR` is repeated *n* times. On UNIX systems, repeat commands can be nested to any level. On OpenVMS systems, `REPEAT` commands can be nested up to three levels and can contain any valid script syntax including delays.

The following example shows nested messages. In this example, the script will send one message of message type 1, four messages of message type 2 with a 1.5-second delay between them, and then send the same messages one more time.

```
REPEAT 2
     MSG
       .
       . ! MSG TYPE 1
       .
     EOM
    REPEAT 4   ! NESTED REPEAT MESSAGE
         MSG
           .
           .   ! MSG TYPE 2
           .
         EOM
         DELAY 1.5
    ENDR
ENDR
```

Entering Time Delays

You can insert a time delay into a script file by using the `DELAY` command. The `DELAY` command allows the simulation of an actual arrival pattern of messages. The `DELAY` command format follows:

`DELAY <`*`min`*`>:<`*`sec`*`>.<`*`tenths`*`>`

`DELAY <`*`min`*`>:<`*`sec`*`>`

`DELAY <`*`sec`*`>.<`*`tenths`*`>`

`DELAY <`*`sec`*`>`

where:
min specifies the number of minutes from 0 to 59; *sec* specifies the number of seconds from 0 to 59; and *tenths* specifies the number of tenths of a second from 0 to 9.

For example, specify a delay of 0.5 second as follows:

```
DELAY 0.5
```

The duration of the delay applies only to the processing of the BEA MessageQ script file and the time of arrival of messages from the BEA MessageQ script file to the user program. The user program will still receive messages from other sources during a delay interval.

Entering Comments

Comments in the script source file can be specified in end-of-line format (`<EOL>`) or comment command format (`COMMENT`).

End-of-Line Format

In the following format, text on the line following the exclamation point (`!`) to the end-of-line tag is ignored. An end-of-line comment can be placed wherever the syntax allows `<EOL>`.

```
! comment text... <EOL>
```

Comment Command Format

In the following format, the text following the `COMMENT` command and all lines within the comment group are ignored until the `ENDC` command terminates the comment. Note that the comment statement can span any number of lines.

```
COMMENT
..this shows comment text
which can span lines..
ENDC
```

Verifying Script Files

Once you have created the script file, you can verify that the syntax is correct before using it. See the following topics for instructions on how to verify scripts on UNIX and OpenVMS systems and how to resolve reported errors.

Verifying Scripts on UNIX Systems

BEA MessageQ for UNIX software provides a utility that verifies script syntax. It is called dmqscript. To verify a script file, use the following command syntax:

```
dmqscript -v -f script_file_name
```

where:

`-v`	Requests verification of script file syntax.
`-f` *`script_file_name`*	Provides the name of the script file to verify. The default extension for script files is `.pss`.

Verifying Scripts on OpenVMS Systems

BEA MessageQ for OpenVMS software provides a utility that verifies script syntax. It is called `DMQ$PSSVFY`. The `DMQ$PSSVFY` utility can be accessed using both a menu interface and a command line interface. To use the menu interface, select the `PSSVFY` option on the BEA MessageQ main menu. You will be prompted to provide the name of the script file to verify.

To use the command line interface, enter the following commands at the DCL prompt:

```
$ PSSVFY :== $ DMQ$EXE:DMQ$PSSVFY
$ PSSVFY script_file_name
```

The default file type for script files is `.PSS`. If you omit the script file name, the utility prompts you to supply it as follows:

```
$_File name:
```

Resolving Script Verification Errors

If the script verification utility does not find any syntax errors in the file, it displays no output on the screen. If errors are found, this command creates a screen display listing syntax errors and the line number on which they were found. Listing 7-8 provides an example of a script file containing errors.

Listing 7-8 Sample Script File with Errors

```
REPEAT 2        !Send this message twice

   MSG SOURCE = 34,1      TARGET = 35,1
       CLASS = MATERIALS
       DX 1, 2, 3
   EOM

   DELAY 10     !Delay 10 seconds before sending the repeat message
```

Listing 7-9 shows the output displayed on the screen when the script containing errors is processed.

Listing 7-9 Sample Output of Script File Verification Utility

```
%PAMSCRIPT-E-IVMSGTARG, Invalid message target name at line 3
-PAMSCRIPT-E-AMBIG, Ambiguous keyword
%PAMSCRIPT-E-NOMSGTO, Missing TARGET phrase in message definition
                      at line 5
%PAMSCRIPT-E-NOMSGTYPE, Missing TYPE phrase in message definition
                      at line 5
%PAMSCRIPT-E-IVDATATYPE, Invalid data type (expecting B, W, or L)
                      at line 5
%PAMSCRIPT-E-MISENDR, Unbalanced REPEAT at line 1, missing closing
                      ENDR command
%PAMSCRIPT-E-ERRORS, Errors encountered in script source file
```

Use the line numbers and error messages to identify the incorrect syntax in your script file. Use a text editor to make the corrections and verify the script again to ensure that all of the errors identified are corrected.

8 PAMS Application Programming Interface

Because the BEA MessageQ application programming interface (API) is portable, the API is documented once for all supported platforms. This chapter describes all BEA MessageQ callable services in alphabetical order using a standard description format.

BEA MessageQ API Description Format

The beginning of each description contains the entry-point name and a brief description of the function performed. Table 8-1 describes the sections in the description of each callable service.

Table 8-1 Callable Service Description Format

In the section entitled . . .	You will find . . .
Syntax	The syntax for using the callable service with the entry-point name and argument list. Square brackets (`[ ]`) indicate optional arguments to the service.
Arguments	The data type, passing mechanism, C language prototype, and access for each argument.
Argument Definitions	Detailed information on how to specify each argument.
Description	More detailed information on how to use the callable service.

Table 8-1 Callable Service Description Format

In the section entitled . . .	You will find . . .
Return Values	The return codes with the platforms on which they are supported.
See Also	A list of related callable services.
Example	A sample program illustrating the use of the callable service. These sample programs are available in the examples directory of the BEA MessageQ media kit.

BEA MessageQ API Data Types

BEA MessageQ API arguments use data types defined by the C programming language and some data types defined by BEA MessageQ software. Data types such as `short`, `unsigned short`, and `char` are data types defined by the C programming language. BEA MessageQ data types such as `q_address` and the PSB and `show_buffer` structures are defined in the `p_entry.h` include file.

BEA MessageQ supports data type definitions for signed and unsigned longwords. The `int32` data type defined by BEA MessageQ is a 32-bit signed integer. The `int32` data type replaces the use of the integer data type long, the size of which is operating system dependent. The `int32` data type definition guarantees a consistent definition across all platforms and was added to accommodate next generation 64-bit architectures such as Compaq's Alpha AXP systems. The `uint32` data type designates a 32-bit unsigned integer and replaces the use of unsigned long.

Note: The `int32` and `uint32` data type definitions are not available on BEA MessageQ Version 2.0 platforms. BEA MessageQ Version 2.0 software uses the standard signed longword and unsigned longword data types defined by the C programming language.

pams_attach_q

Connects an application program to the BEA MessageQ message queuing bus by attaching it to a message queue. An application must successfully execute this function before it can send and receive messages. When an application uses this function to attach to a queue, it becomes the owner of the queue. Only one application program can attach to a primary queue and read messages from it. When an application attaches to a permanent primary queue defined with secondary queue attachments, the secondary queues are also attached by this function.

Syntax

```
int32 pams_attach_q ( attach_mode, q_attached, [q_type], [q_name],
[q_name_len], [name_space_list], [name_space_list_len], [timeout],
[nullarg_2], [nullarg_3] )
```

Arguments

Table 8-2

Argument	Data Type	Mechanism	Prototype	Access
`attach_mode`	`int32`	reference	`int32 *`	passed
`q_attached`	`q_address`	reference	`q_address *`	returned
`[q_type]`	`int32`	reference	`int32 *`	passed
`[q_name]`	`char`	reference	`char *`	passed
`[q_name_len]`	`int32`	reference	`int32 *`	passed
`[name_space_list]`	`int32 array`	reference	`int32 array *`	passed
`[name_space_list_len]`	`int32`	reference	`int32 *`	passed
`[timeout]`	`int32`	reference	`int32 *`	passed
`[nullarg_2]`	`char`	reference	`char *`	passed
`[nullarg_3]`	`char`	reference	`char *`	passed

Argument Definitions

attach_mode

Supplies the mode for attaching the application to a message queue. The three predefined constants for this argument are:

- PSYM_ATTACH_BY_NAME—Attach by name
- PSYM_ATTACH_BY_NUMBER—Attach by number
- PSYM_ATTACH_TEMPORARY—Attach as a temporary queue

When **attach_mode** is PSYM_ATTACH_BY_NAME, the application attaches to the queue identified by the specified queue or alias name. BEA MessageQ finds the queue by implicitly performing a pams_locate_q call for the specified **q_name**. The procedure that BEA MessageQ uses is determined by the **name_space_list** argument.

q_attached

Receives the queue address from BEA MessageQ when an application has successfully attached to a message queue.

q_type

Supplies the queue type for the attachment. The two predefined constants for this argument are:

- PSYM_ATTACH_PQ—Primary queue (default)
- PSYM_ATTACH_SQ—Secondary queue

q_name

Supplies the name or number of the permanent queue to attach to the application if the **attach_mode** argument specifies attachment by queue name or queue number. Queue names are alphanumeric strings with no embedded spaces and allow the following special characters: underscore (_), hyphen (-), and dollar sign ($).

References to queue names are case sensitive and must match the queue name entered in the group initialization file. Some example queue names are: QUEUE_1, high-priority, and My$Queue.

The **q_name** argument has the following dependencies with the **attach_mode** argument:

- If the **attach_mode** argument is PSYM_ATTACH_BY_NAME, the **q_name** argument must contain a valid queue name as specified during BEA MessageQ group configuration.
- If the **attach_mode** argument is PSYM_ATTACH_BY_NUMBER, the **q_name** argument is specified as an ASCII string of 1 to 3 numeric characters representing the queue number.

- If the **attach_mode** argument is PSYM_ATTACH_TEMPORARY, the **q_name** argument is not used and should be specified by passing a value of 0.

q_name_len

Supplies the number of characters in the **q_name** argument. The maximum string length on UNIX, Windows NT, and OpenVMS servers is 255 characters. For all other BEA MessageQ environments, the maximum string length is 31.

name_space_list

Supplies a list of name tables to search when the **attach_mode** argument PSYM_ATTACH_BY_NAME is specified.

If the **name_space_list** is specified, then the **name_space_list_len** argument must also be specified. If this argument is unspecified, then PSEL_TBL_GRP is the default.

Possible values in a **name_space_list** argument are as follows:

Location It Represents	Symbolic Value
Process cache	PSEL_TBL_PROC
Group/group cache	PSEL_TBL_GRP
Global name space	PSEL_TBL_BUS (or PSEL_TBL_BUS_MEDIUM or PSEL_TBL_BUS_LOW)

The **name_space_list** argument identifies the scope of the name as follows:

- To identify a local queue reference or a queue, an application must include PSEL_TBL_GRP in **name_space_list.** (Do not specify PSEL_TBL_BUS in the list because it would identify a global queue reference.)
- To identify a global queue reference, include PSEL_TBL_BUS (or PSEL_TBL_BUS_MEDIUM or PSEL_TBL_BUS_LOW) in the **name_space_list** argument and specify its pathname, either explicitly or implicitly. If the **q_name** argument contains any slashes (/), or periods (.), BEA MessageQ treats it as a pathname. Otherwise, BEA MessageQ treats **q_name** as a name and adds the DEFAULT_NAMESPACE_PATH to the name to create the pathname to lookup. (The

DEFAULT_NAMESPACE_PATH is set in the %PROFILE section of the group initialization file.)

The **name_space_list** argument also controls the cache access as follows.

- To cause the lookup of a local queue reference or queue name to check the process cache before looking in the group cache, specify the **name_space_list** argument as PSEL_TBL_GRP and PSEL_TBL_PROC.
- To cause the lookup of a global queue reference to check the process cache and then the group cache before looking into the global name space, specify PSEL_TBL_BUS(or PSEL_TBL_BUS_LOW or PSEL_TBL_BUS_MEDIUM), PSEL_TBL_GRP and PSEL_TBL_PROC.

  To lookup all caches in the global name space before looking in the master database, specify PSEL_TBL_BUS_LOW instead of PSEL_TBL_BUS.

  To lookup only the slower but more up-to-date caches in the global name space before looking in the master database, specify PSEL_TBL_BUS_MEDIUM instead of PSEL_TBL_BUS.

For more information on dynamic binding of queue addresses, see the Using Naming topic.

name_space_list_len

Supplies the number of entries in the **name_space_list** argument. If the **name_space_list_len** argument is zero, BEA MessageQ uses PSEL_TBL_GRP as the default in the **name_space_list** argument.

timeout

The number of PAMS time units (1/10 second intervals) to allow for the attach to complete. If a zero is specified, the group's ATTACH_TMO property is used. If the ATTACH_TMO property is also zero, 600 is used.

nullarg_2

Reserved for BEA MessageQ internal use as a placeholder argument. This argument must be supplied as a null pointer.

nullarg_3

Reserved for BEA MessageQ internal use as a placeholder argument. This argument must be supplied as a null pointer.

Description

Before an application can use the `pams_attach_q` function, the BEA MessageQ **message queuing bus** must be configured. A BEA MessageQ message queuing bus is a collection of one or more BEA MessageQ **message queuing groups**. A message queuing group is a collection of *message queues* that reside on a system, share global memory sections and files, and are served by the same server processes. A BEA MessageQ message queue is an area of memory or disk where messages are stored and retrieved. See the installation and configuration guide for the platform you are using to learn how to configure the BEA MessageQ environment.

To receive BEA MessageQ messages, an application must attach to at least one message queue. The `pams_attach_q` function enables an application to attach in the following ways:

- An application can attach to a queue by specifying a **number**. To attach by number, the message queue must be configured in the group definition. Attaching by number enables an application to attach to a specific queue, send messages to the queue, and retrieve messages sent to that queue.
- An application can attach to a queue by specifying the queue **name**. To attach by name, the message queue must be configured in the group definition. Attaching by name enables an application to attach to a specific queue, send messages to the queue, and retrieve messages sent to that queue. In addition, attaching by name eliminates the need to change code or recompile if the queue address changes. Therefore, attaching by name protects applications from changes in the BEA MessageQ environment configuration.
- An application can attach to a **temporary** queue. To attach to a temporary queue, the application does not have to give a specific queue name or number. BEA MessageQ will assign a queue and return the number of the queue which has been assigned. Temporary queues allow an application to perform messaging without knowing configuration details of the group.

Applications can specify an attachment as primary or secondary. All applications must have a primary queue. In addition, applications can attach to one or more secondary queues. Primary queues can be configured in the group definition as the owners of secondary queues. When an application attaches to a primary queue that is the owner of secondary queues, the application is automatically attached to the secondary queues at the same time it is attached to the primary queue.

In addition, an application can attach to a multireader queue. A multireader queue can be read by many applications and is configured as part of the group definition.

Return Values

Table 8-3

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Wrong number of call arguments has been passed to this function.
`PAMS__BADDECLARE`	All	Queue has already been attached to this application.
`PAMS__BADNAME`	All	Invalid name string was specified.
`PAMS__BADPARAM`	All	Invalid argument in the argument list.
`PAMS__BADPROCNUM`	All	Queue number out of range.
`PAMS__BADQTYPE`	All	Invalid queue type.
`PAMS__BADTMPPROC`	OpenVMS	Invalid temporary queue number.
`PAMS__DECLARED`	All	The queue number is already attached to another application or process.
`PAMS__DUPLQNAME`	OpenVMS	Duplicate queue name.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOACCESS`	All	No access to the resource. The address of the specified name is either 0 or it is in another group.
`PAMS__NOACL`	All	The queue access control file could not be found.
`PAMS__NOOBJECT`	All	No such queue name. For a global queue reference, this error can be caused by a bad default pathname in the group configuration file.
`PAMS__NOQUOTA`	OpenVMS	Insufficient receive message or byte quota to attach.
`PAMS__NOTBOUND`	All	The queue name is not bound to an address.

Table 8-3

Return Code	Platform	Description
`PAMS__NOTMRQ`	OpenVMS	Attempting to attach to Multi-reader Queue and queue type is not an MRQ.
`PAMS__NOTPRIMARYQ`	All	Queue name or number is not a primary queue.
`PAMS__NOTSECONDARYQ`	All	Queue name or number is not a secondary queue.
`PAMS__PAMSDOWN`	All	The specified BEA MessageQ group is not running.
`PAMS__PREVCALLBUSY`	Clients	The previous call to CLS has not been completed.
`PAMS__PNUMNOEXIST`	OpenVMS	Target queue name or number does not exist.
`PAMS__RESRCFAIL`	All	Failed to allocate resources.
`PAMS__SUCCESS`	All	Successful completion of an action.
`PAMS__TIMEOUT`	All	The timeout period specified has expired.

See Also

- `pams_detach_q`
- `pams_exit`
- `pams_locate_q`

Examples

- **Attach by Name**—this example illustrates how to attach to a queue by name. The name “`example_q_1`” must be defined in your group configuration information as a primary queue or as a local queue alias or a primary queue. The complete code example called `x_attnam.c` is contained in the examples directory.
- **Attach by Number**—this example illustrates how to attach to a queue by number. A queue numbered 1 must be defined in your group configuration information file as a primary queue. The complete code example called `x_attnum.c` is contained in the examples directory.

- **Attach as Temporary**—this example illustrates how to attach as a temporary queue. The complete code example called `x_atttmp.c` is contained in the examples directory.

pams_bind_q

Dynamically associates a queue address to a queue reference at run-time. This enables a server application to dynamically sign up to service a queue alias at run-time. Thus, an end user can access a service without having to be aware that its normal host computer is down and that the service is being provided from another host computer.

Syntax

```
int32 pams_bind_q (q_addr, q_alias, q_alias_len, [name_space_list],
                  [name_space_list_len], [timeout], [nullarg_1]);
```

Arguments

Table 8-4

Argument	Data Type	Mechanism	Prototype	Access
`q_addr`	`q_address`	reference	`q_address *`	passed
`q_alias`	`char`	reference	`char *`	passed
`q_alias_len`	`int32`	reference	`int32 *`	passed
`[name_space_list]`	`int32 array`	reference	`int32 array *`	passed
`[name_space_list_len]`	`int32`	reference	`int32 *`	passed
`[timeout]`	`int32`	reference	`int32 *`	passed
`[nullarg_1]`	`char`	reference	`char *`	passed

Argument Definitions

q_addr

The value specified to this argument controls whether the queue address is bound or unbound:

- If the queue address is specified, this function binds it to a **`q_alias`**.
- If 0 is specified, this function unbinds the **`q_alias`** from its queue address. The calling application must be bound to **`q_alias`** to set it back to zero.

q_alias

Identifies a global queue reference or a local queue reference. The procedure that BEA MessageQ uses to find this alias is controlled by the **`name_space_list`** argument, which is described below.

q_alias_len

Specifies the number of characters in **`q_alias`**.

name_space_list

If specified, identifies a one-entry list containing either `PSEL_TBL_BUS` or `PSEL_TBL_GRP`.

To identify a local queue reference, an application must have a name space list of `PSEL_TBL_GRP` and pass its name in the **`q_alias`** argument. To identify a global queue reference, an application must have a name space list of `PSEL_TBL_BUS` and specify its pathname, either explicitly or implicitly:

- If the **`q_alias`** argument contains any slashes (`/`), or periods (`.`), BEA MessageQ treats the **`q_alias`** as a pathname.
- Otherwise, BEA MessageQ treats **`q_alias`** as a name and adds the group's `DEFAULT_NAMESPACE_PATH` to the name to create the pathname to lookup. (The `DEFAULT_NAMESPACE_PATH` is set in the `%PROFILE` section of the initialization file.)

For more information on dynamic binding of queue addresses, see the Using Naming topic.

name_space_list_len

Specifies the number of entries in **`name_space_list`** argument. The number of entries is either 0 or 1. If the number of entries is 0 (indicating that the **`name_space_list`** is omitted), `PSEL_TBL_GRP` is assumed.

timeout

Specifies the number of PAMS time units (1/10 second intervals) to allow for the bind to complete. If 0 is specified, the group's `ATTACH_TMO` property is used. If the `ATTACH_TMO` property is also 0, 600 is used.

nullarg_1

Reserved for BEA MessageQ internal use as a placeholder argument. This argument must be supplied as a null pointer.

Description

Before an application can call `pams_bind_q`, it must be attached to the specified queue address. Listing 8-1 shows an attach before the bind call and is typical usage of the two functions together:

Listing 8-1 Example of Using pams_bind_q

```
int32 mode = PSYM_ATTACH_BY_NUMBER;
             int32 q_type = PSYM_ATTACH_PQ;
             int32 len=1;
             int32 status;
             q_address qid;


status = pams_attach_q(&mode,&qid,&q_type,"2",&len,0,0,0,0,0);

         if (status == PAMS__SUCCESS {
             int32 ns=PSEL_TBL_BUS;
             int32 ns_len=1;
             len = strlen("Q2");

status = pams_bind_q(&qid,"Q2",&len,&ns,&ns_len,0,0);
                }
```

Return Values

Table 8-5

Return Code	Platform	Description
`PAMS__BADARGLIST`	All	Invalid number of call arguments.
`PAMS__BADNAME`	All	Name contains bad characters.
`PAMS__BADPARAM`	All	The name space list is invalid.
`PAMS__BOUND`	All	Returned if a non-zero value for **`q_addr`** is passed and the specified **`q_alias`** is already assigned to a queue address.

Table 8-5

Return Code	Platform	Description
`PAMS__DUPLQNAME`	All	Duplicate queue name.
`PAMS__FAIL`	All	Operation failed.
`PAMS__NOACCESS`	All	No access to the resource. The address of the specified name is either 0 or it is in another group.
`PAMS__NOOBJECT`	All	For a global reference, this error can be caused by a bad default pathname in the group configuration file.
`PAMS__NOTBOUND`	All	The queue name is not bound to an address.
`PAMS__NOTDCL`	All	Not attached to BEA MessageQ.
`PAMS__PAMSDOWN`	All	The specified BEA MessageQ group is not running.
`PAMS__SUCCESS`	All	Indicates successful completion.
`PAMS__TIMEOUT`	All	The timeout period specified has expired. In this situation, BEA MessageQ internally unbinds the specified queue alias. Subsequent `pams_bind_q` calls to the same name will return the `PAMS__UNBINDING` error until the internal unbind succeeds.
`PAMS__UNBINDING`	All	The bind cannot be done because BEA MessageQ is still in the process of has unbinding the old binding.

See Also

- `pams_attach_q`
- `pams_locate_q`

Example

The `pams_bind_q` example illustrates how to bind a queue reference to a queue address at runtime. The complete code example called `x_bind.c` is contained in the examples directory.

pams_cancel_get

Cancels all pending pams_get_msga requests that match the value specified in the **sel_filter** argument. When a pending pams_get_msga request is canceled, the PAMS Status Block (PSB) delivery status is set to PAMS__CANCEL and the specified action routine is queued. The pams_cancel_get function waits until completion to allow for proper synchronization between the pams_cancel_get function and the request for pams_get_msga functions. Any outstanding pams_get_msga function requests are canceled by the pams_exit function or at image exit.

Syntax

```
int32 pams_cancel_get ( sel_filter )
```

Arguments

Table 8-6

Argument	Data Type	Mechanism	Prototype	Access
sel_filter	int32	reference	int32 *	passed

Argument Definition

sel_filter

Supplies the criteria that enables the application to selectively cancel outstanding pams_get_msga requests. For a description of the **sel_filter** argument, see the pams_get_msg function. For a description of how to create a complex selection filter, see the pams_set_select function.

Return Values

Table 8-7

Return Code	Platform	Description
PAMS__BADARGLIST	OpenVMS	Argument list is invalid.
PAMS__SUCCESS	OpenVMS	Indicates successful completion.
SS$_EXQUOTA	OpenVMS	Process has exceeded its asynchronous system trap (AST) quota.

See Also

- pams_cancel_select
- pams_get_msga
- pams_set_select

pams_cancel_select

Releases the selection array and index handle associated with a previously generated selection mask. An **`index_handle`** and associated selection mask are created using the `pams_set_select` function. When the selection mask is used in the OpenVMS environment with asynchronous read requests, this function also cancels any pending `pams_get_msga` requests that use the referenced **`index_handle`**.

Syntax

```
int32 pams_cancel_select ( index_handle )
```

Arguments

Table 8-8

Argument	Data Type	Mechanism	Prototype	Access
`index_handle`	`int32`	reference	`int32 *`	passed

Argument Definitions

index_handle

Supplies the index handle of the selection mask to cancel. The **`index_handle`** is returned by the `pams_set_select` function.

Return Values

Table 8-9

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Invalid number of call arguments.
`PAMS__BADPARAM`	UNIX Windows NT	The value of the selection index is null.
`PAMS__BADSELIDX`	All	Invalid or undefined selective receive index.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOTDCL`	All	Process has not been attached to BEA MessageQ.
`PAMS__PAMSDOWN`	UNIX Windows NT	The specified BEA MessageQ group is not running.

Table 8-9

Return Code	Platform	Description
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__SUCCESS`	All	Indicates successful completion.

See Also

- `pams_get_msga`
- `pams_set_select`

pams_cancel_timer

Deletes the BEA MessageQ timer identified by the **`timer_id`** argument that is passed to this function. All expired timers with the selected identification code that are waiting in the message queue are purged and are not delivered.

Syntax

```
int32 pams_cancel_timer ( timer_id )
```

Arguments

Table 8-10

Argument	Data Type	Mechanism	Prototype	Access
`timer_id`	`int32`	reference	`int32 *`	passed

Argument Definitions

timer_id

Supplies the timer ID of the timer to cancel. The **`timer_id`** is returned by the `pams_set_timer` function.

Return Values

Table 8-11

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Invalid number of arguments.
`PAMS__BADPARAM`	All	The **`timer_id`** argument was specified as null.
`PAMS__INVALIDNUM`	All	The application has supplied an invalid value for the **`timer_id`**.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOTDCL`	All	The application has not attached to a queue.
`PAMS__PAMSDOWN`	UNIX Windows NT	The specified BEA MessageQ group is not running.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.

Table 8-11

Return Code	Platform	Description
`PAMS__RESRCFAIL`	All	Insufficient resources to complete the operation.
`PAMS__SUCCESS`	All	Indicates successful completion.

See Also

- `pams_set_timer`

pams_close_jrn

Closes the MRS journal file associated with the **`jrn_handle`** argument. The two types of journal files are dead letter journal (DLJ) and postconfirmation journal (PCJ). See Using Recoverable Messaging for a description of how to use the BEA MessageQ message recovery system.

Syntax

```
int32 pams_close_jrn ( jrn_handle )
```

Arguments

Table 8-12

Argument	Data Type	Mechanism	Prototype	Access
`jrn_handle`	`int32`	reference	`int32 *`	passed

Argument Definitions

Jrn_handle

Supplies the journal handle of the message recovery journal file to close. The **`jrn_handle`** is returned by the `pams_open_jrn` function.

Return Values

Table 8-13

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Invalid number of arguments.
`PAMS__INVJH`	OpenVMS	The application has supplied an invalid journal handle.
`PAMS__SUCCESS`	OpenVMS	Indicates successful completion.

See Also

- `pams_confirm_msg`
- `pams_open_jrn`
- `pams_read_jrn`

pams_confirm_msg

Confirms receipt of a message that requires explicit confirmation. This can be a recoverable message sent to a queue that is configured for explicit confirmation or a message sent using the ACK delivery mode which must be explicitly confirmed upon receipt. Applications should examine the PSB status field of each message received to determine if the message requires explicit confirmation.

When a recoverable message is received, the application must call the `pams_confirm_msg` function in order to delete it from the message recovery journal disk storage. If receipt of a recoverable message is not confirmed, the message continues to be stored by the recovery system and will be redelivered if the application detaches and then reattaches to the queue.

BEA MessageQ can confirm receipt of a recoverable message automatically when the next consecutive message in the recovery journal is delivered. This feature is called implicit confirmation.

All queues must be configured for implicit or explicit confirmation. For complete information on how to configure message queues, see the installation and configuration guide for your system.

Successfully delivered recoverable messages can be recorded in the postconfirmation journal (PCJ). The `pams_confirm_msg` function uses the **`force_j`** argument to write messages to the PCJ file if the system is not currently configured to store them. Note that successfully delivered recoverable messages cannot be written to the PCJ file unless they are explicitly confirmed using the `pams_confirm_msg` function.

Syntax

```
int32 pams_confirm_msg ( msg_seq_num, confirmation_status,
            force_j )
```

Arguments

Table 8-14

Argument	Data Type	Mechanism	Prototype	Access
`msg_seq_num`	`uint32 array`	reference	`uint32 array *`	passed
`confirmation_status`	`int32`	reference	`int32 *`	passed
`force_j`	`char`	reference	`char *`	passed

Argument Definitions

msg_seq_num

Supplies the message sequence number of the recoverable message being confirmed. The message sequence number is generated by the BEA MessageQ message recovery system for each recoverable message. This value is passed to the receiver program in the PSB of the `pams_get_msg` function when it reads each recoverable message.

confirmation_status

Supplies the confirmation status value stored with the message in the postconfirmation journal (PCJ) file. The value is set by the calling application. See the Using Recoverable Messaging topic for more information on using the PCJ file.

force_j

Supplies the journaling action for this message. Following are the predefined constants for this argument:

Symbol	Description
`PDEL_DEFAULT_JRN`	Enables writing the message to the PCJ file if the journaling is configured in the group initialization file.
`PDEL_FORCE_JRN`	Enables writing to the PCJ only if journaling is configured. It is not possible to write messages to the PCJ on UNIX and Windows NT systems if a path was not defined for the PCJ in the group configuration information.
`PDEL_NO_JRN`	Disables journaling regardless of whether journaling is configured.

Description

The PSB status codes associated with recoverable message delivery are

`PAMS__CONFIRMREQ` and `PAMS__POSSDUPL`. The `PAMS__CONFIRMREQ` PSB status code indicates that it is the first time the application received the recoverable message. The `PAMS__POSSDUPL` status code indicates that the message was retrieved from the recovery journal and may have been sent previously. This status code allows the application to take extra precautions to handle duplicate messages if necessary.

The PSB also contains a sequence number that uniquely identifies the message. The `pams_confirm_msg` function requires this sequence number. If one of these status codes is present and the `pams_confirm_msg` function is not called, the message will continue to be stored by the message recovery system and will be delivered again if the application exits and then reattaches.

Return Values

Table 8-15

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Invalid number of arguments.
`PAMS__BADPARAM`	All	Bad argument value.
`PAMS__BADSEQ`	All	Journal sequence number is not known to the Message Recovery Services (MRS).
`PAMS__DQF_DEVICE_FAIL`	OpenVMS	I/O error writing to the destination queue file for the target queue.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOMRS`	All	MRS is not available.
`PAMS__NOTDCL`	All	Process is not attached to BEA MessageQ.
`PAMS__NOTJRN`	All	Message is not written to the PCJ file.
`PAMS__NOTSUPPORTED`	OpenVMS	Attached to the dead letter queue.
`PAMS__PAMSDOWN`	UNIX Windows NT	The specified BEA MessageQ group is not running.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__RESRCFAIL`	OpenVMS	BEA MessageQ resources exhausted.
`PAMS__SUCCESS`	All	Indicates successful completion.

See Also

- `pams_get_msg`

- pams_get_msga
- pams_get_msgw
- pams_put_msg

Example **Confirm Receipt of Recoverable Messages**

This example demonstrates using recoverable messaging. It attaches to queue_1, puts some recoverable messages to queue_2, exits, attaches to queue_2, gets the messages, prints them out, then exits.

The queues named “queue_1” and “queue_2” are defined in your initialization file. On OpenVMS systems, you must set up a DQF for queue_2. The complete code example called x_recovr.c is contained in the examples directory.

pams_detach_q

Detaches a selected message queue or all of the application's message queues from the message queuing bus. When an application detaches from its primary queue, this function automatically detaches all secondary queue attachments defined for the primary queue. When the last message queue has been detached, the application is automatically detached from the BEA MessageQ message queuing bus.

Syntax

```
int32 pams_detach_q ( q, detach_opt_list, detach_opt_len,
                      msgs_flushed )
```

Arguments

Table 8-16

Argument	Data Type	Mechanism	Prototype	Access
`q`	`q_address`	reference	`q_address *`	passed
`detach_opt_list`	`int32 array`	reference	`int32 *`	passed
`detach_opt_len`	`int32`	reference	`int32 *`	passed
`msg_flushed`	`int32`	reference	`int32 *`	returned

Argument Definitions

q

Supplies the queue address of the queue to be detached. This function can be used to detach primary, secondary, and multireader queues.

detach_opt_list

Supplies an array of `int32` values used to control how the queue is detached. The predefined constants for this argument are:

- `PSYM_NOFLUSH_Q`—Detaches the queue without flushing the pending messages stored in memory. The default action is to flush pending messages in the queue before it is detached. Messages are never flushed from multireader queues.
- `PSYM_DETACH_ALL`—Detaches all of the application's message queues from the message queuing bus. Using this constant performs the same action as calling the `pams_exit` function.

- PSYM_CANCEL_SEL_MASK—Cancels all selection masks that reference the queue or queues that you are detaching. If you do not select this option and you do not cancel selection masks, BEA MessageQ invalidates all selection masks that reference the queue or queues that you are detaching. You must cancel the invalidated selection masks using the pams_cancel_select function.

detach_opt_len

Supplies the number of int32 values in the **detach_opt_list** array. The maximum number of int32 longwords is 32,767.

msgs_flushed

Receives the number of messages that were flushed from the queue. Message count statistics are enabled on all systems by default; therefore, it is not necessary to enable statistics on UNIX and Windows NT systems in order to properly return this value.

Description

If you are using implicit confirmation with recoverable messaging, you must ensure that the last message is confirmed before:

- Detaching from the queue which received the message by calling pams_detach_q
- Detaching from the message queuing bus by calling pams_exit
- Exiting your application

If you do not ensure that the last message was confirmed before detaching or exiting, the message will be redelivered when the queue is reattached. The easiest method to ensure confirmation is to save the PSB delivery status of the last message received, check it for the required confirmation status, and then exit after the message has been confirmed.

Return Values

Table 8-17

Return Code	Platform	Description
PAMS__BADARGLIST	OpenVMS	Invalid number of arguments.
PAMS__BADPARAM	All	Invalid **detach_opt_list**.
PAMS__DETACHED	All	Process has detached from BEA MessageQ.

Table 8-17

Return Code	Platform	Description
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOTDCL`	All	Not attached to BEA MessageQ.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__PNUMNOEXIST`	All	Invalid queue address or queue not owned by process.
`PAMS__SUCCESS`	All	Queue successfully detached.

See Also

- `pams_attach_q`
- `pams_exit`

pams_exit

Terminates all attachments between the application and the BEA MessageQ message queuing bus. All pending messages in temporary queues and permanent queues which are not permanently active multi-reader queues are discarded. Only the messages pending in permanently active multi-reader queues are retained. To retain messages in permanently active queues, call `pams_detach_q` with option `PSYM_NOFLUSH_Q` before calling `pams_exit`.

Syntax

```
int32 pams_exit (void)
```

Arguments

None.

Description

If you are using implicit confirmation with recoverable messaging, you must ensure that the last message is confirmed before:

- Detaching from the queue which received the message by calling `pams_detach_q`
- Detaching from the message queuing bus by calling `pams_exit`
- Exiting your application

If you do not ensure that the last message was confirmed before detaching or exiting, the message will be redelivered when the queue is reattached. The easiest method to ensure confirmation is to save the PSB delivery status of the last message received, check it for the required confirmation status, and then exit after the message has been confirmed.

Return Values

Table 8-18

Return Code	Platform	Description
`PAMS__NETERROR`	OpenVMS Client	Network error resulted in a communications link abort.
`PAMS__NOTDCL`	OpenVMS	Not attached to BEA MessageQ.
`PAMS__PREVCALLBUSY`	OpenVMS Client	Previous call to CLS has not been completed.
`PAMS__PNUMNOEXIST`	OpenVMS	Invalid queue address or queue not owned by process.

Table 8-18

Return Code	Platform	Description
`PAMS__SUCCESS`	All	Indicates successful completion.

See Also

- `pams_attach_q`
- `pams_detach_q`

Example

Exit the Message Queuing Bus

This example shows how to use the `pams_exit` function. The complete code example called `x_exit.c` is contained in the examples directory.

pams_get_msg

Retrieves the next available message from a selected queue and moves it to the location specified in the **`msg_area`** argument. When no selection filter is specified, the function returns the next available message in first-in/first-out (FIFO) order based on message priority to the buffer specified in the **`msg_area`** argument. Priority ranges from 0 (lowest priority) to 99 (highest priority). For example, priority 1 messages are always placed before priority 0 messages. Messages are placed in first-in/first out order by message priority. If a selection filter is specified, then only messages that meet the selection criteria are retrieved. If no messages are available or meet the selection criteria, then the return status is `PAMS__NOMOREMSG`.

Applications should check the PSB status field of each message to determine if the message was sent with a recoverable delivery mode. If an application receives a recoverable message, it must call the `pams_confirm_msg` function to delete it from the message recovery journal disk storage. If receipt of a recoverable message is not confirmed, the message continues to be stored by the recovery system and will be redelivered if the application detaches and then reattaches to the queue.

The receiver program determines whether each message is a FML32 buffer or large message by reading the **`msg_area_len`** argument. See the Sending and Receiving BEA MessageQ Messages topic for more information on working with FML32 buffers and large messages.

Syntax

```
int32 pams_get_msg ( msg_area, priority, source, class, type,
                     msg_area_len, len_data, [sel_filter], [psb],
                     [show_buffer], [show_buffer_len], [large_area_len],
                     [large_size], [nullarg_3] )
```

Arguments

Table 8-19

Argument	Data Type	Mechanism	Prototype	Access
`msg_area`	`char`	reference	`char *`	returned
`priority`	`char`	reference	`char *`	passed
`source`	`q_address`	reference	`q_address *`	returned
`class`	`short`	reference	`short *`	returned
`type`	`short`	reference	`short *`	returned

Table 8-19

Argument	Data Type	Mechanism	Prototype	Access
`msg_area_len`	`short`	reference	`short *`	passed
`len_data`	`short`	reference	`short*`	returned
`[sel_filter]`	`int32`	reference	`int32 *`	passed
`[psb]`	`struct psb`	reference	`struct psb *`	returned
`[show_buffer]`	`struct show_buffer`	reference	`struct show_buffer *`	returned
`[show_buffer_len]`	`int32`	reference	`int32 *`	passed
`[large_area_len]`	`int32`	reference	`int32 *`	passed/ returned
`[large_size]`	`int32`	reference	`int32 *`	returned
`[nullarg_3]`	`char`	reference	`char*`	passed

Argument Definitions

msg_area

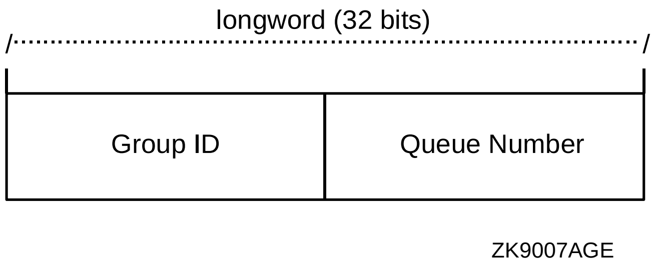
For static buffer-style messaging, receives the address of a memory region where BEA MessageQ writes the contents of the retrieved message. For FML-style messaging or when using double pointers, receives a pointer to the address of the message being retrieved.

priority

Supplies the priority level for selective message reception. Priority ranges from 0 (lowest priority) to 99 (highest priority). If the priority is set to 0, the `pams_get_msqw` function gets messages of any priority. If the priority is set to any value from 1 to 99, the `pams_get_msqw` function gets only messages of that priority.

source

Receives a data structure containing the group ID and queue number of the sender program’s primary queue in the following format:



class

Receives the class code of the retrieved message. The class is specified in the `pams_put_msg` function. BEA MessageQ supports the use of symbolic names for class argument values. Symbolic class names should begin with `MSG_CLAS_`. For information on defining class symbols, see the `p_typecl.h` include file. On UNIX and Windows NT systems, the `p_typecl.h` include file cannot be edited. You must create an include file to define type and class symbols for use by your application.

Class symbols reserved by BEA MessageQ are as follows:

Reserved Class	Symbol Value
`MSG_CLAS_MRS`	28
`MSG_CLAS_PAMS`	29
`MSG_CLAS_ETHERNET`	100
`MSG_CLAS_UCB`	102
`MSG_CLAS_TUXEDO`	31001
`MSG_CLAS_TUXEDO_TPSUCCESS`	31002
`MSG_CLAS_TUXEDO_TPFAIL`	31003
`MSG_CLAS_XXX`	30000 through 32767 (except 31001-31003)

type

Receives the type code of the retrieved message. The type is specified in the `pams_put_msg` function. BEA MessageQ supports the use of symbolic names for **`type`** argument values. Symbolic type names begin with `MSG_TYPE_`. For specific information on defining type symbols, see the `p_typecl.h` include file.

BEA MessageQ has reserved the symbol value range -1 through -5000. A zero value for this argument indicates that no processing by message type is expected.

msg_area_len

- Supplies the size of the buffer (in bytes) for static message buffers of up to 32767 bytes. The **`msg_area`** buffer is used to store the retrieved message.
- For messages using double buffers, including FML32 buffers, this argument contains the symbol `PSYM_MSG_BUFFER_PTR` to indicate that the message is a pointer to the address of the message being retrieved. The **`msg_area`** buffer contains the message pointer. The size of the message is returned in the **`large_size`** argument. The **`msg_area`** buffer is used to store the retrieved message. The **`large_area_len`** argument is used to supply the size of the message buffer to receive the message. If the retrieved buffer is larger than the space allocated, space is dynamically reallocated and the new buffer size is stored in **`large_area_len`**.
- For large messages (buffer-style messages larger than 32767 bytes), this argument contains the symbol `PSYM_MSG_LARGE` to indicate that the message buffer is greater than 32K. The size of the message is returned in the **`large_size`** argument. The **`msg_area`** buffer is used to store the retrieved message. The **`large_area_len`** argument is used to supply the size of the message buffer to receive the large message.

len_data

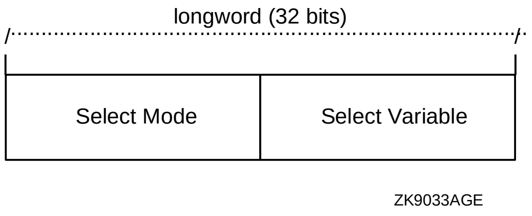
For static buffer-style messaging with messages of up to 32767 bytes, this argument receives the number of bytes retrieved from the message queue and stored in the area specified by the **`msg_area`** argument. This field also receives the `PSYM_MSG_BUFFER_PTR` symbol for double buffer and FML-style messages and `PSYM_MSG_LARGE` for buffer-style messages larger than 32767 bytes.

sel_filter

Supplies the criterion to enable the application to selectively retrieve messages. The argument contains one of the following selection criteria:

- Default selection
- Selection by message queue
- Message attributes
- Message source
- Compound selection using the `pams_set_select` function

The **`sel_filter`** argument is composed of two words as follows:



Default Selection

Enables applications to read messages from the queue based on the order in which they arrived. The default selection, `PSEL_DEFAULT`, reads the next pending message from the message queue. Messages are stored by priority and then in FIFO order. To specify this explicitly, both words in the **`sel_filter`** argument should be set to 0.

Selection by Message Queue

Allows the application to retrieve messages based upon a queue type or combination of queue types. This selection criteria is used to retrieve the first pending message that matches the criteria on the first queue it encounters. FIFO ordering is maintained within each queue. The predefined constants for this argument are as follows:

Table 8-20

Select Mode	Select Variable	Mode Description
`PSEL_PQ`	0	Enables the application to read from the primary queue (PQ) only. The select variable must equal 0.
`PSEL_AQ`	Alternate queue number	Enables an application to read from an alternate queue (AQ) only. The queue type can be a secondary queue (SQ).

Table 8-20

Select Mode	Select Variable	Mode Description
`PSEL_PQ_AQ`	Alternate queue number	Attempts to selectively retrieve from a primary queue and then from an alternate queue.
`PSEL_AQ_PQ`	Alternate queue number	Attempts to selectively retrieve from an alternate queue and then from a primary queue.
`PSEL_TQ_PQ`	Alternate queue number	Attempts to selectively retrieve messages from a timer queue (TQ), and then from a primary queue.
`PSEL_TQ_PQ_AQ`	Alternate queue number	Attempts to selectively retrieve messages from a timer queue (TQ), then from a primary queue, and finally from an alternate queue.
`PSEL_UCB`	0	Retrieves messages only from the user callback queues (UCB).

Selection by Message Attribute

Enables the application to select messages based on the message type, class, or priority. The predefined constants for this argument are as follows:

Table 8-21

Select Mode	Select Variable	Mode Description
`PSEL_PQ_TYPE`	Type	Selects the first pending message from the primary queue that matches the type value in the select variable word.
`PSEL_PQ_CLASS`	Class	Selects the first pending message from the primary queue that matches the class value in the select variable word.

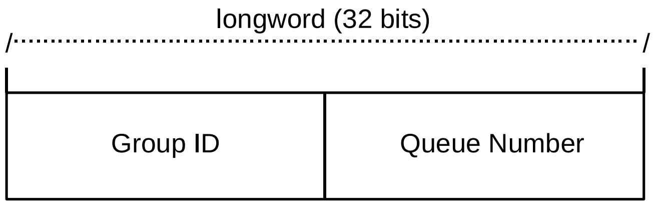
Table 8-21

Select Mode	Select Variable	Mode Description

`PSEL_PQ_PRI`	`PSEL_PRI_ANY`
	`PSEL_PRI_P0`
	`PSEL_PRI_P1`
	integer value between 0 and 99
		Selects the first pending message with a priority equal to an integer between 0 and 99 inclusive (or equal to the select variable value) from within the primary queue. Specifying the direct integer value is the preferred method of selected messages by priority.
		Using PSEL_PRI_ANY enables the reading of any pending messages of all priorities. Setting PSEL_PRI_P0 enables the application to retrieve pending messages of priority 0 only. Setting PSEL_PRI_P1 enables the strict retrieval of pending messages with a priority of 1.

Selection by Message Source

Provides for the selection of pending messages from primary and secondary queues, by source group ID, queue number, or both. The format for selection by source follows:



ZK9007AGE

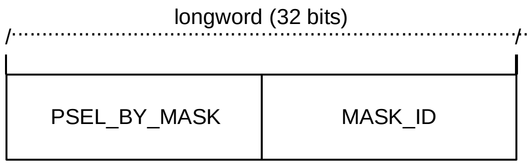
Some examples of possible **`sel_filter`** arguments and their actions are as follows:

sel_filter Argument	Action
Zero or not specified	No filtering of any messages. All messages can be retrieved.
Source **`q_address`**	Only those messages that have a matching **`q_address`** are retrieved.

sel_filter Argument	**Action**
Selection mask created with `pams_set_select`	Only messages that exactly match the specified selection mask are retrieved.

Compound Selection

Allows the application to formulate complex rules for the order in which the message queues are searched. The `pams_set_select` function allows the application to create custom selection masks that can be used in the low-order word of the **`sel_filter`** argument. The format for compound selection follows:



ZK9034AGE

psb

Receives a PAMS Status Block containing the final completion status. The **`psb`** argument is used when sending or receiving recoverable messages. The PSB structure stores the status information from the message recovery system and may be checked after sending or receiving a message. The structure of the PSB is as follows:

Table 8-22

Low Byte	**High Byte**	**Contents**	**Description**
0	1	Type	PSB type
2	3	Call Dependent	Currently not used.
4	7	PSB Delivery Status	The completion status of the function. For recoverable messages, this field contains `PAMS__CONFIRMREQ` or `PAMS__POSSDUPL`. For nonrecoverable messages, it may also contain a value of `PAMS__SUCCESS`.

Table 8-22

Low Byte	High Byte	Contents	Description
8	15	Message Sequence Number	A unique number assigned to a message when it is sent and follows the message to the destination PSB. This number is input to `pams_confirm_msg` to release a recoverable message.
16	19	PSB UMA Status	This field is not used for the `pams_get_msg` function.
20	23	Function Return Status	This field is not used for the `pams_get_msg` function.
24	31	Not Used	Not used.

show_buffer

Receives additional information which BEA MessageQ extracts from the message header. The structure of the **`show_buffer`** argument is as follows:

Table 8-23

Longword	Contents	Description
0	Version	The version of the show_buffer structure. Valid values are as follows: `10` = Version 1.0 `20` = Version 2.0 `50` = Version 5.0
1	Transfer Status	The status code associated with the transfer of **`show_buffer`** information into the application's buffer. Valid symbols are as follows: `PAMS__SUCCESS`—All available information has been transferred. `PAMS__BUFFEROVF`—Information was lost due to receiver buffer overflow. 0—No message returned. There is no information to transfer.

Table 8-23

Longword	Contents	Description
2	Transfer Size	The number of bytes transferred to the application buffer.

3	Flags	A bit array showing the status of fields in the `show_buffer`. A set bit indicates a valid field, while a cleared bit indicates indeterminable data or the end of the allocated `show_buffer` memory. The symbols for the flags field are as follows:
		`PSYM_SHOW_VERSION`
		`PSYM_SHOW_STATUS`
		`PSYM_SHOW_SIZE`
		`PSYM_SHOW_FLAGS`
		`PSYM_SHOW_TARGET`
		`PSYM_SHOW_ORIGINAL_TARGET`
		`PSYM_SHOW_SOURCE`
		`PSYM_SHOW_ORIGINAL_SOURCE`
		`PSYM_SHOW_DELIVERY`
		`PSYM_SHOW_PRIORITY`
		`PSYM_SHOW_ENDIAN`
		`PSYM_SHOW_CORRELATION_ID`

4	Not Used	Fills out the Control Section to its maximum 24 bytes.
5	Not Used	Fills out the Control Section to its maximum 24 bytes.
6	Not Used	Fills out the Control Section to its maximum 24 bytes.
7	Not Used	Fills out the Control Section to its maximum 24 bytes.
8	Not Used	Fills out the Control Section to its maximum 24 bytes.
9	Not Used	Fills out the Control Section to its maximum 24 bytes.
10	Target	The **`q_address`** of the latest message target.
11	Original Target	The **`q_address`** of the original message target.
12	Source	The **`q_address`** of the latest message source.
13	Original Source	The **`q_address`** of the original message.
14	Delivery Mode	The delivery mode that was used to queue the message. This is not necessarily the delivery mode used to generate the message.

Table 8-23

Longword	Contents	Description
15	Priority	The priority used to queue the message.

16	Endian	The byte ordering or encoding schemes of 2- and 4-byte integers. The possible settings are as follows:
		`PSYM_UNKNOWN`
		`PSYM_VAX_BYTE_ORDER` or `PSYM_LITTLE_ENDIAN`
		`PSYM_NETWORK_BYTE_ORDER` or
		`PSYM_BIG_ENDIAN`
		`PSYM_FML`

17	Correlation ID	The 32 byte correlation ID associated with the message.

show_buffer_len

Supplies the length in bytes of the buffer defined in the **`show_buffer`** argument. The minimum length is 40 bytes. If the buffer is too small to contain all of the information, then the return code `PAMS_BUFFEROVF` will be in the **`show_buffer`** transfer status.

large_area_len

Specifies the size of the message area to receive messages larger than 32K. Also specifies the length of the message buffer when using double buffers (as indicated by **`PSYM_MSG_BUFFER_POINTER`**). This argument also stores the length of double buffers and FML32 buffers after reallocation.

large_size

Returns the actual size of the large message, double buffer message, or FML32 message written to the message buffer.

nullarg_3

Reserved for BEA MessageQ internal use as a placeholder argument. This argument must be supplied as a null pointer.

Return Values

Table 8-24

Return Code	Platform	Description
`PAMS__AREATOSMALL`	All	Received message is larger than the user's message area.
`PAMS__BADARGLIST`	All	Wrong number of call arguments have been passed to this function.
`PAMS__BADHANDLE`	All	Invalid message handle.
`PAMS__BADPARAM`	All	Bad argument value.
`PAMS__BADPRIORITY`	All	Invalid priority value used for receive.
`PAMS__BADSELIDX`	All	Invalid or undefined selective receive index.
`PAMS__BUFFEROVF`	UNIX Windows NT	The size of the `show_buffer` specified is too small.
`PAMS__EXHAUSTBLKS`	OpenVMS	No more message blocks available.
`PAMS__FMLERROR`	All	Problem detected with internal format of FML message; this can be an error in processing or data corruption.
`PAMS__INSQUEFAIL`	All	Failed to properly queue a message buffer.
`PAMS__MSGTOSMALL`	All	The **`msg_area_len`** argument must be positive or zero.
`PAMS__MSGUNDEL`	All	Message returned is undeliverable.
`PAMS__NEED_BUFFER_PTR`	UNIX Windows NT	FML32 buffer received but **`msg_area_len`** argument not set to `PSYM_MSG_BUFFER_PTR`.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOACCESS`	All	No access to resource.
`PAMS__NOACL`	All	Queue access control file could not be found.

Table 8-24

Return Code	Platform	Description
`PAMS__NOMEMORY`	OpenVMS	Insuffucient memory resources to reallocate buffer pointer.
`PAMS__NOMOREMSG`	All	No messages available.
`PAMS__NOMRQRESRC`	All	Insufficient multireader queue resources to allow access.
`PAMS__NOTDCL`	All	Process has not been attached to BEA MessageQ.
`PAMS__NOTSUPPORTED`	UNIX Windows NT	The supplied delivery mode is not supported.
`PAMS__PAMSDOWN`	UNIX Windows NT	The specified BEA MessageQ group is not running.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__QUECORRUPT`	OpenVMS	Message buffer queue corrupt.
`PAMS__REMQUEFAIL`	All	Failed to properly read from a message buffer.
`PAMS__STALE`	All	Resource is no longer valid and must be freed by the user.
`PAMS__STOPPED`	All	The requested queue has been stopped.
`PAMS__SUCCESS`	All	Indicates successful completion.

PBS Delivery Status

Table 8-25

PSB Delivery Status	Platform	Description
`PAMS__CONFIRMREQ`	All	Confirmation required for this message.
`PAMS__PAMSDOWN`	UNIX Windows NT	The specified BEA MessageQ group is not running.

Table 8-25

PSB Delivery Status	Platform	Description
`PAMS__POSSDUPL`	All	Message is a possible duplicate.
`PAMS__SUCCESS`	All	Indicates successful completion.

See Also

- `pams_get_msga`
- `pams_get_msgw`
- `pams_put_msg`
- `pams_set_select`

Example **Read a Message**

This example uses the `pams_get_msg` function to retrieve all the messages currently in the queue and sends them to a print function. The complete code example called `x_get.c` is contained in the examples directory.

pams_get_msga

The pams_get_msga function is only available on OpenVMS systems.

Requests asynchronous notification of a message arrival. The pams_get_msga function triggers an asynchronous system trap (AST) routine when a message arrives in that queue. Notification to the application occurs by triggering an AST, by setting an event flag, or both.

When no selection filter is specified, the function returns the next available message in first-in/first-out (FIFO) order based on message priority to the user-supplied **msg_area** argument. Priority ranges from 0 (lowest priority) to 99 (highest priority). For example, priority 1 messages are always placed before priority 0 messages. Messages are placed in first-in/first out order by message priority. If a selection filter is specified, then only messages that meet the selection criteria are retrieved, and the AST or event flag is triggered only when a matching message arrives.

If a queue has been sent a recoverable message, the receiver program can confirm receipt of the message using the pams_confirm_msg function. The pams_confirm_msg function enables the successfully delivered message to be deleted from the message recovery system. See the Using Recoverable Messaging topic for a description of the BEA MessageQ recovery system.

See the Sending and Receiving BEA MessageQ Messages topic for more information on working with FML32 buffers and large messages.

Syntax

```
int32 pams_get_msga ( msg_area, priority, source, class, type,
               msg_area_len, len_data, [sel_filter], [psb],
               [show_buffer], [show_buffer_len],
               [large_area_len], [large_size], [actrtn],
               [actparm], [flag_id], [nullarg_3] )
```

Arguments

Table 8-26

Argument	Data Type	Mechanism	Prototype	Access
msg_area	char	reference	char *	returned
priority	char	reference	char *	passed
source	q_address	reference	q_address *	returned
class	short	reference	short *	returned

Table 8-26

Argument	Data Type	Mechanism	Prototype	Access
`type`	`short`	reference	`short *`	returned
`msg_area_len`	`short`	reference	`short *`	passed
`len_data`	`short`	reference	`short *`	returned
`[sel_filter]`	`int32`	reference	`int32 *`	passed
`[psb]`	`struct psb`	reference	`struct psb *`	returned
`[show_buffer]`	`struct show_buffer`	reference	`struct show_buffer *`	returned
`[show_buffer_len]`	`int32`	reference	`int32 *`	passed
`[large_area_len]`	`int32`	reference	`int32 *`	passed/ returned
`[large_size`	`int32`	reference	`int32 *`	returned
`[actrtn]`	`int32`	value	`int32 *`	passed
`[actparm]`	`int32`	reference	`int32 *`	passed
`[flag_id]`	`int32`	reference	`int32 *`	passed
`[nullarg_3]`	`char`	reference	`char *`	passed

Argument Definitions

msg_area

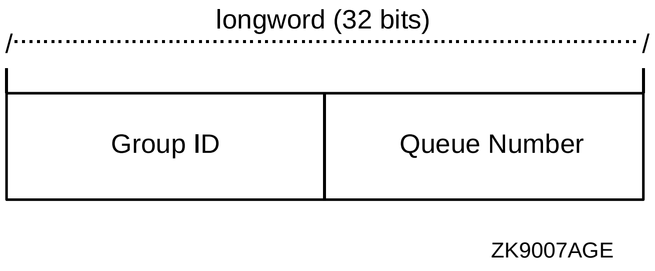
For static buffer-style messaging, receives the address of a memory region where BEA MessageQ writes the contents of the retrieved message. For FML-style messaging or when using double pointers, receives a pointer to the address of the message being retrieved. When using double buffer pointers with `pams_get_msga`, the new buffer size is returned in `large_size`. (This differs from `pams_get_msg[w}`, where the new buffer size is returned in `large_area_len`.)

priority

Supplies the priority level for selective message reception. Priority ranges from 0 (lowest priority) to 99 (highest priority)..

source

Receives a data structure containing the group ID and queue number of the sender program's primary queue in the following format:



class

Receives the class code of the retrieved message. The class is specified in the `pams_put_msg` function. BEA MessageQ supports the use of symbolic names for **`class`** argument values. Symbolic class names should begin with `MSG_CLAS_`. For information on defining class symbols, see the `p_typecl.h` include file.

Class symbols reserved by BEA MessageQ are as follows:

Reserved Class	Symbol Value
`MSG_CLAS_MRS`	28
`MSG_CLAS_PAMS`	29
`MSG_CLAS_ETHERNET`	100
`MSG_CLAS_UCB`	102
`MSG_CLAS_TUXEDO`	31001
`MSG_CLAS_TUXEDO_TPSUCCESS`	31002
`MSG_CLAS_TUXEDO_TPFAIL`	31003
`MSG_CLAS_XXX`	30000 through 32767 (except 31001-31003)

type

Receives the type code of the retrieved message. The type is specified in the `pams_put_msg` function. BEA MessageQ supports the use of symbolic names for **`type`** argument values. Symbolic type names begin with `MSG_TYPE_`. For specific information on defining type symbols, see the `p_typecl.h` include file.

BEA MessageQ has reserved the symbol value range –1 through –5000. A zero value for this argument indicates that no processing by message type is expected.

msg_area_len

- Supplies the size of the buffer (in bytes) for buffer-style messages of up to 32767 bytes. The **`msg_area`** buffer is used to store the retrieved message.
- For messages using double buffers, including FML32 buffers, this argument contains the symbol `PSYM_MSG_BUFFER_PTR` to indicate that the message is a pointer to the address of the message being retrieved. The **`msg_area`** buffer contains the message pointer. The size of the message is returned in the **`large_size`** argument. The **`msg_area`** buffer is used to store the retrieved message. The **`large_area_len`** argument is used to supply the size of the message buffer to receive the message. If the retrieved buffer is larger than the space allocated, space is dynamically reallocated and the new buffer size is stored in **`large_size`**.
- For large messages (buffer-style messages larger than 32767 bytes), this argument contains the symbol `PSYM_MSG_LARGE` to indicate that the message buffer is greater than 32K. The size of the message is returned in the **`large_size`** argument The **`msg_area`** buffer is used to store the retrieved message. The **`large_area_len`** argument is used to supply the size of the message buffer to receive the large message.

len_data

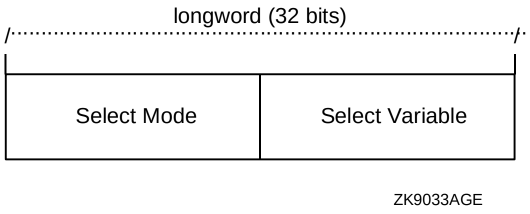
For static buffer-style messaging with messages of up to 32767 bytes, this argument receives the number of bytes retrieved from the message queue and stored in the area specified by the **`msg_area`** argument. This field also receives the `PSYM_MSG_BUFFER_PTR` symbol for FML-style messages and `PSYM_MSG_LARGE` for buffer-style messages larger than 32767 bytes.

sel_filter

Supplies the criteria enabling the application to selectively retrieve messages. The argument contains one of the following selection criteria:

- Default selection
- Selection by message queue
- Message attributes
- Message source
- Compound selection using the `pams_set_select` function

The **`sel_filter`** argument is composed of two words as follows:



Default Selection Enables applications to read messages from the queue based on the order in which they arrived. The default selection, `PSEL_DEFAULT`, reads the next pending message from the message queue. Messages are stored by priority and then in FIFO order. To specify this explicitly, both words in the **`sel_filter`** argument should be set to 0.

Selection by Message Queue Allows the application to retrieve messages based upon a queue type or combination of queue types. This selection criteria is used to retrieve the first pending message that matches the criteria on the first queue it encounters. FIFO ordering is maintained within each queue.

The predefined constants for this argument are as follows:

Table 8-27

Select Mode	Select Variable	Mode Description
`PSEL_PQ`	0	Enables the application to read from the primary queue (PQ) only. The select variable must equal 0.
`PSEL_AQ`	Alternate queue number	Enables an application to read from an alternate queue (AQ) only. The queue type can be a secondary queue (SQ).

Table 8-27

Select Mode	Select Variable	Mode Description
`PSEL_PQ_AQ`	Alternate queue number	Attempts to selectively retrieve from a primary queue and then from an alternate queue.
`PSEL_AQ_PQ`	Alternate queue number	Attempts to selectively retrieve from an alternate queue and then from a primary queue.
`PSEL_TQ_PQ`	Alternate queue number	Attempts to selectively retrieve messages from a timer queue (TQ), and then from a primary queue.
`PSEL_TQ_PQ_AQ`	Alternate queue number	Attempts to selectively retrieve messages from a timer queue (TQ), then from a primary queue, and finally from an alternate queue.
`PSEL_UCB`	0	Retrieves messages only from the user callback queues (UCB).

Selection by Message Attribute

Enables the application to select messages based on the message type, class, or priority. The predefined constants for this argument are as follows:

Table 8-28

Select Mode	Select Variable	Mode Description
`PSEL_PQ_TYPE`	Type	Selects the first pending message from the primary queue that matches the type value in the select variable word.
`PSEL_PQ_CLASS`	Class	Selects the first pending message from the primary queue that matches the class value in the select variable word.

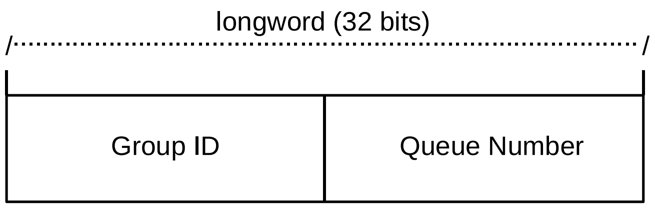
Table 8-28

Select Mode	Select Variable	Mode Description

`PSEL_PQ_PRI`	`PSEL_PRI_ANY`
	`PSEL_PRI_P0`
	`PSEL_PRI_P1`
	integer value between 0 and 99
		Selects the first pending message with a priority equal to an integer between 0 and 99 inclusive (or equal to the select variable value) from within the primary queue. Specifying the direct integer value is the preferred method of selected messages by priority.
		Using PSEL_PRI_ANY enables the reading of any pending messages of all priorities. Setting PSEL_PRI_P0 enables the application to retrieve pending messages of priority 0 only. Setting PSEL_PRI_P1 enables the strict retrieval of pending messages with a priority of 1.

Selection by Message Source

Provides for the selection of pending messages from primary and secondary queues, by source group ID, queue number, or both. The format for selection by source follows:



ZK9007AGE

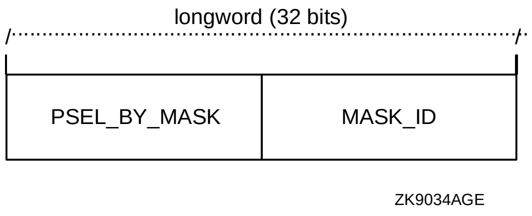
Some examples of possible **`sel_filter`** arguments and their actions are as follows:

sel_filter Argument	Action
Zero or not specified	No filtering of any messages. All messages can be retrieved.
Source **`q_address`**	Only those messages that have a matching **`q_address`** are retrieved.

sel_filter Argument	**Action**
Selection mask created with `pams_set_select`	Only messages that exactly match the specified selection mask are retrieved.

Compound Selection

Allows the application to formulate complex rules for the order in which the message queues are searched. The `pams_set_select` function allows the application to create custom selection masks that can be used in the low-order word of the **`sel_filter`** argument. The format for compound selection follows.



psb

Receives a PAMS Status Block containing the final completion status. The **`psb`** argument is used when sending or receiving recoverable messages. The PSB structure stores the status information from the message recovery system and may be checked after sending or receiving a message. The structure of the PSB is as follows:

Table 8-29

Low Byte	**High Byte**	**Contents**	**Description**
0	1	Type	PSB type
2	3	Call Dependent	Currently not used.
4	7	PSB Delivery Status	The completion status of the function. For recoverable messages, this field contains `PAMS__CONFIRMREQ` or `PAMS__POSSDUPL`. For nonrecoverable messages, it may also contain a value of `PAMS__SUCCESS`.

Table 8-29

Low Byte	High Byte	Contents	Description
8	15	Message Sequence Number	A unique number assigned to a message when it is sent and follows the message to the destination PSB. This number is input to `pams_confirm_msg` to release a recoverable message.
16	19	PSB UMA Status	This field is not used with the `pams_get_msga` function.
20	23	Function Return Status	This field is not used with the `pams_get_msga` function.
24	31	Not Used	Not used.

Note: This function utilizes the AST services of OpenVMS; therefore, the application must check the status information returned in the PSB.

show_buffer

Receives additional information which BEA MessageQ extracts from the message header. The structure of the **`show_buffer`** argument is as follows:

Table 8-30

Longword	Contents	Description
0	Version	The version of the **`show_buffer`** structure. Valid values are as follows: 10 = Version 1.0 20 = Version 2.0 50 = Version 5.0

Table 8-30

Longword	Contents	Description
1	Transfer Status	The status code associated with the transfer of **`show_buffer`** information into the application's buffer. Valid symbols are as follows: `PAMS__SUCCESS`—All available information has been transferred. `PAMS__BUFFEROVF`—Information was lost due to receiver buffer overflow. 0—No message returned. There is no information to transfer.
2	Transfer Size	The number of bytes transferred to the application buffer.

3	Flags	A bit array showing the status of fields in the `show_buffer`. A set bit indicates a valid field, while a cleared bit indicates indeterminable data or the end of the allocated `show_buffer` memory. The symbols for the flags field are as follows:
		`PSYM_SHOW_VERSION`
		`PSYM_SHOW_STATUS`
		`PSYM_SHOW_SIZE`
		`PSYM_SHOW_FLAGS`
		`PSYM_SHOW_TARGET`
		`PSYM_SHOW_ORIGINAL_TARGET`
		`PSYM_SHOW_SOURCE`
		`PSYM_SHOW_ORIGINAL_SOURCE`
		`PSYM_SHOW_DELIVERY`
		`PSYM_SHOW_PRIORITY`
		`PSYM_SHOW_ENDIAN`
		`PSYM_SHOW_CORRELATION_ID`

4	Not Used	Fills out the Control Section to its maximum 40 bytes.
5	Not Used	Fills out the Control Section to its maximum 40 bytes.
6	Not Used	Fills out the Control Section to its maximum 40 bytes.
7	Not Used	Fills out the Control Section to its maximum 40 bytes.
8	Not Used	Fills out the Control Section to its maximum 40 bytes.
9	Not Used	Fills out the Control Section to its maximum 40 bytes.

Table 8-30

Longword	Contents	Description
10	Target	The **q_address** of the latest message target.
11	Original Target	The **q_address** of the original message target.
12	Source	The **q_address** of the latest message source.
13	Original Source	The **q_address** of the original message.
14	Delivery Mode	The delivery mode that was used to queue the message. This is not necessarily the delivery mode used to generate the message.
15	Priority	The priority used to queue the message.
16	Endian	The byte ordering or encoding schemes of 2- and 4-byte integers. The possible settings are as follows: PSYM_UNKNOWN PSYM_VAX_BYTE_ORDER or PSYM_LITTLE_ENDIAN PSYM_NETWORK_BYTE_ORDER or PSYM_BIG_ENDIAN PSYM_FML
17	Correlation ID	The 32 byte correlation ID associated with the message.

show_buff_len

Supplies the length in bytes of the buffer defined in the **show_buffer** argument. The minimum length is 40 bytes. If the buffer is too small to contain all of the information, then the return code PAMS__BUFFEROVF will be in the **show_buffer** transfer status.

large_area_len

Specifies the size of the message area to receive messages larger than 32K. Also specifies the length of the message buffer when using double buffers (as indicated by **PSYM_MSG_BUFFER_POINTER**).

large_size

Returns the actual size of the large message, double buffer message, or FML32 message written to the message buffer. When using double buffer pointers with `pams_get_msga`, the new buffer size is returned in `large_size`. (This differs from `pams_get_msg[w}`, where the new buffer size is returned in `large_area_len`.)

actrtn

Supplies the address of an int32 value that is the entry point to an action routine. This action routine is executed when the `pams_get_msga` function completes.

actparm

Supplies an int32 value that is passed to the action routine specified in the **`actrtn`** argument when it is invoked.

flag_id

Supplies the `int32` value for the flag number to be set when the `pams_get_msga` function completes. When the `pams_get_msga` function executes, it clears this flag. If this argument value is not supplied, no flag is used.

nullarg_3

Reserved for BEA MessageQ internal use as a placeholder argument. This argument must be supplied as a null pointer.

Description

Because the `pams_get_msga` function executes asynchronously, it obtains several argument values only after the message arrives. These argument values are the message buffer, source, class, type of the message, and a PAMS Status Block (PSB) status code containing the delivery status, UMA status, and the sequence number of the message. These values are not set until the message arrival triggers the AST routine or sets the event flag.

The `pams_get_msga` function specifies an AST parameter which is passed by value to the AST routine when the parameter is called. This parameter is used to provide a context for the information contained in the message and can be used to identify the specific processing required for the message. Following are some suggestions and rules for programming with ASTs:

- Create a context area, separate from mainline, for each AST that is simultaneously posted. An address or index associated with the context area should be used as the AST parameter to ensure the appropriate context is associated with the data that is delivered by the `pams_get_msga` function.

- Ensure that the addresses of any fields that are filled in asynchronously are valid throughout the period that the AST is posted. A common error in using ASTs is to post an AST request that fills in fields on the stack and becomes invalid as soon as the caller returns.
- Data may be passed between AST routines and mainline by the following mechanisms:
  - BEA MessageQ messages.
  - An event queue managed by interlocked queuing instructions.
  - Shared data fields between mainline and the AST routines such that access to the data is clear. The use of a context area for each AST request can accomplish this.
- Access to complex data structures shared between mainline and AST routines should be serialized by placing the access inside an AST safe critical section. One way to do this is with the `$SETAST` system service.

Return Values

Table 8-31

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Wrong number of call arguments have been passed to this function.
`PAMS__BADPARAM`	OpenVMS	Bad argument value.
`PAMS__BADPRIORITY`	OpenVMS	Invalid priority value used for receive.
`PAMS__BADSELIDX`	OpenVMS	Invalid or undefined selective receive index.
`PAMS__BADHANDLE`	OpenVMS	Invalid message handle.
`PAMS__MSGTOSMALL`	OpenVMS	The **`msg_area_len`** argument must be positive or zero.
`PAMS__NOACCESS`	OpenVMS	No access to the queue.
`PAMS__NOACL`	OpenVMS	No access to resource. The ACL check failed.

Table 8-31

Return Code	Platform	Description
PAMS__NOMEMORY	OpenVMS	Insuffucient memory resources to reallocate buffer pointer.
PAMS__NOTDCL	OpenVMS	The application has not been attached to BEA MessageQ.
PAMS__NOTSUPPORTED	OpenVMS	Feature not supported or available.
PAMS__RESRCFAIL	OpenVMS	Failed to allocate a resource.
PAMS__STALE	OpenVMS	Resource is no longer valid and must be freed by the user.
PAMS__STOPPED	OpenVMS	The requested queue has been stopped.
PAMS__SUCCESS	OpenVMS	Indicates successful completion.

PSB Delivery Status

Table 8-32

PSB Delivery Status	Platform	Description
PAMS__CONFIRMREQ	OpenVMS	Confirmation required for this message.
PAMS__POSSDUPL	OpenVMS	Message is a possible duplicate.
PAMS__SUCCESS	OpenVMS	Indicates successful completion.

See Also

- `pams_cancel_get`
- `pams_get_msg`
- `pams_get_msgw`
- `pams_put_msg`
- `pams_set_select`

pams_get_msgw

Retrieves the next available message from a specified queue and moves it to the location specified in the **`msg_area`** argument. This function waits until a message arrives in the queue or a user-specified timeout period has elapsed.

When no selection filter is specified, the function returns the next available message in first-in/first-out (FIFO) order based on message priority to the user-supplied **`msg_area`** argument. Priority ranges from 0 (lowest priority) to 99 (highest priority). If the priority is set to 0, the `pams_get_msqw` function gets messages of any priority. If the priority is set to any value from 1 to 99, the `pams_get_msqw` function gets only messages of that priority. Messages are placed in first-in/first-out order by message priority. If a selection filter is specified, then only messages that meet the selection criteria are retrieved. If no message arrives, or no available message meets the selection criteria before the **`timeout`** period expires, then the return status is `PAMS__TIMEOUT`.

If a queue has been sent a recoverable message, the receiver program can confirm receipt of the message using the `pams_confirm_msg` function. The `pams_confirm_msg` function enables the successfully delivered message to be deleted from the message recovery system. See the Using Recoverable Messaging topic for a description of the BEA MessageQ recovery system.

See the Sending and Receiving BEA MessageQ Messages topic for more information on working with FML32 buffers and large messages.

Syntax

```
int32 pams_get_msgw ( msg_area, priority, source, class, type,
                      msg_area_len, len_data, timeout, [sel_filter],
                      [psb], [show_buffer], [show_buffer_len],
                      [large_area_len], [large_size],[nullarg_3] )
```

Argument

Table 8-33

Argument	Data Type	Mechanism	Prototype	Access
`msg_area`	char	reference	char *	returned
`priority`	char	reference	char *	passed
`source`	`q_address`	reference	`q_address *`	returned
`class`	`short`	reference	`short *`	returned

Table 8-33

Argument	Data Type	Mechanism	Prototype	Access
`type`	`short`	reference	`short *`	returned
`msg_area_len`	`short`	reference	`short *`	passed
`len_data`	`short`	reference	`short *`	returned
`timeout`	`int32`	reference	`int32 *`	passed
`[sel_filter]`	`int32`	reference	`int32 *`	passed
`[psb]`	`struct psb`	reference	`struct psb *`	returned
`[show_buffer]`	`struct show_buffer`	reference	`struct show_buffer *`	returned
`[show_buffer_len]`	`int32`	reference	`int32 *`	passed
`[large_area_len]`	`int32`	reference	`int32 *`	passed/ returned
`[large_size]`	`int32`	reference	`int32 *`	returned
`[nullarg_3]`	`char`	reference	`char *`	passed

Argument Definitions

msg_area

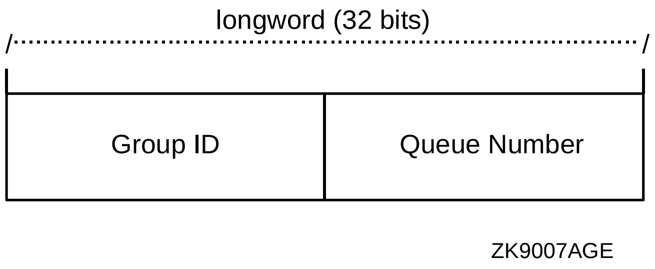
For buffer-style messaging, receives the address of a memory region where BEA MessageQ writes the contents of the retrieved message. For FML-style messaging or when using double ponters, receives a pointer to the address of the message being retrieved.

priority

Supplies the priority level for selective message reception. Priority ranges from 0 (lowest priority) to 99 (highest priority). If the priority is set to 0, the `pams_get_msqw` function gets messages of any priority. If the priority is set to any value from 1 to 99, the `pams_get_msqw` function gets only messages of that priority.

source

Receives a structure identifying the group ID and queue number of the sender program's primary queue in the following format:



class

Receives the class code of the retrieved message. The class is specified in the arguments of the `pams_put_msg` function. BEA MessageQ supports the use of symbolic names for **`class`** argument values. Symbolic class names should begin with `MSG_CLAS_`. For information on defining class symbols, see the `p_typecl.h` include file. On UNIX and Windows NT systems, the `p_typecl.h` include file cannot be edited. You must create an include file to define type and class symbols for use by your application.

Class symbols reserved by BEA MessageQ are as follows:

Reserved Class	Symbol Value
`MSG_CLAS_MRS`	28
`MSG_CLAS_PAMS`	29
`MSG_CLAS_ETHERNET`	100
`MSG_CLAS_UCB`	102
`MSG_CLAS_TUXEDO`	31001
`MSG_CLAS_TUXEDO_TPSUCCESS`	31002
`MSG_CLAS_TUXEDO_TPFAIL`	31003
`MSG_CLAS_XXX`	30000 through 32767 (except 31001-31003)

type

Receives the type code of the retrieved message. The type is specified in the arguments of the `pams_put_msg` function. BEA MessageQ supports the use of symbolic names for **`type`** argument values. Symbolic type names begin with `MSG_TYPE_`. For specific information on defining type symbols, see the `p_typecl.h` include file.

BEA MessageQ has reserved the symbol value range -1 through -5000. A zero value for this argument indicates that no processing by message type is expected.

msg_area_len

- Supplies the size of the buffer (in bytes) for buffer-style messages of up to 32767 bytes. The **`msg_area`** buffer is used to store the retrieved message.
- For messages using double buffers, including FML32 buffers, this argument contains the symbol `PSYM_MSG_BUFFER_PTR` to indicate that the message is a pointer to the address of the message being retrieved. The **`msg_area`** buffer contains the message pointer. The size of the message is returned in the **`large_size`** argument. The **`msg_area`** buffer is used to store the retrieved message. The **`large_area_len`** argument is used to supply the size of the message buffer to receive the message. If the retrieved buffer is larger than the space allocated, space is dynamically reallocated and the new buffer size is stored in **`large_area_len`**.
- For large messages (buffer-style messages larger than 32767 bytes), this argument contains the symbol `PSYM_MSG_LARGE` to indicate that the message buffer is greater than 32K. The size of the message is returned in the **`large_size`** argument. The **`msg_area`** buffer is used to store the retrieved message. The **`large_area_len`** argument is used to supply the size of the message buffer to receive the large message.

len_data

For static buffer-style messaging with messages of up to 32767 bytes, this argument receives the number of bytes retrieved from the message queue and stored in the area specified by the **`msg_area`** argument. This field also receives the `PSYM_MSG_BUFFER_PTR` symbol for double buffer and FML-style messages and `PSYM_MSG_LARGE` for buffer-style messages larger than 32767 bytes.

timeout

Supplies the maximum amount of time the `pams_get_msg` function waits for a message to arrive before returning control to the application. The timeout value is entered in tenths (0.1) of a second. A value of 100 indicates a timeout of 10 seconds. If the timeout occurs before a message arrives, the status code of `PAMS__TIMEOUT` is returned.

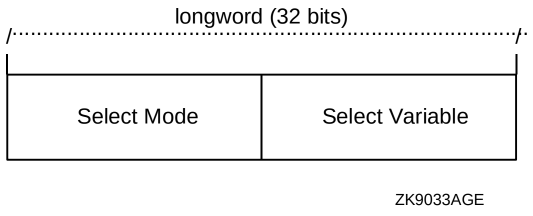
If an unlimited timeout period is required, set this argument to 0. On UNIX and Windows NT systems, a value of zero for this argument causes this function to block indefinitely or until it receives a message. On OpenVMS systems, this function waits for approximately 5 days or until it receives a message.

sel_filter

Supplies the criteria for the application to selectively retrieve messages. The argument contains one of the following selection criteria:

- Default selection
- Selection by message queue
- Message attributes
- Message source
- Compound selection using the `pams_set_select` function

The **`sel_filter`** argument is composed of two words as follows:



Default Selection

Enables applications to read messages from the queue based on the order in which they arrived. The default selection, `PSEL_DEFAULT`, reads the next pending message from the message queue. Messages are stored by priority and then in FIFO order. To specify this explicitly, both words in the **`sel_filter`** argument should be set to 0.

Selection by Message Queue

Allows the application to retrieve messages based upon a queue type or combination of queue types. This selection criteria is used to retrieve the first pending message that matches the criteria on the first queue it encounters. FIFO ordering is maintained within each queue. The predefined constants for this argument are as follows:

Table 8-34

Select Mode	Select Variable	Mode Description
`PSEL_PQ`	0	Enables the application to read from the primary queue (PQ) only. The select variable must equal 0.
`PSEL_AQ`	Alternate queue number	Enables an application to read from an alternate queue (AQ) only. The queue type can be a secondary queue (SQ).
`PSEL_PQ_AQ`	Alternate queue number	Attempts to selectively retrieve from a primary queue and then from an alternate queue.
`PSEL_AQ_PQ`	Alternate queue number	Attempts to selectively retrieve from an alternate queue and then from a primary queue.
`PSEL_TQ_PQ`	Alternate queue number	Attempts to selectively retrieve messages from a timer queue (TQ), and then from a primary queue.
`PSEL_TQ_PQ_AQ`	Alternate queue number	Attempts to selectively retrieve messages from a timer queue (TQ), then from a primary queue, and finally from an alternate queue.
`PSEL_UCB`	0	Retrieves messages only from the user callback queues (UCB).

Selection by Message Attribute

Enables the application to select messages based on the message type, class, or priority. The predefined constants for this argument are as follows:

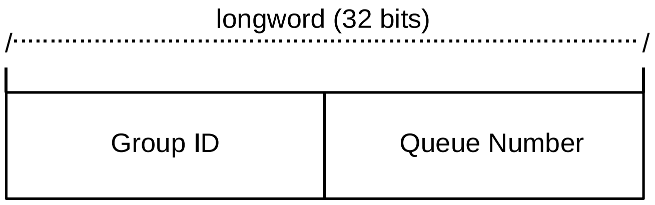
Table 8-35

Select Mode	Select Variable	Mode Description
`PSEL_PQ_TYPE`	Type	Selects the first pending message from the primary queue that matches the type value in the select variable word.
`PSEL_PQ_CLASS`	Class	Selects the first pending message from the primary queue that matches the class value in the select variable word.

`PSEL_PQ_PRI`	`PSEL_PRI_ANY`
	`PSEL_PRI_P0`
	`PSEL_PRI_P1`
	integer value between 0 and 99
		Selects the first pending message with a priority equal to an integer between 0 and 99 inclusive (or equal to the select variable value) from within the primary queue. Specifying the direct integer value is the preferred method of selected messages by priority.
		Using `PSEL_PRI_ANY` enables the reading of any pending messages of all priorities. Setting `PSEL_PRI_P0` enables the application to retrieve pending messages of priority 0 only. Setting `PSEL_PRI_P1` enables the strict retrieval of pending messages with a priority of 1.

Selection by Message Source

Provides for the selection of pending messages from primary and secondary queues, by source group ID, queue number, or both. The format for selection by source follows:



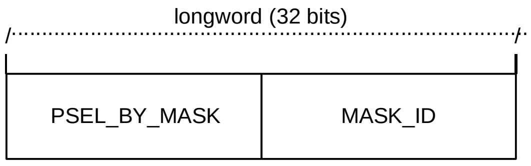
ZK9007AGE

Some examples of possible **`sel_filter`** arguments and their actions are as follows:

sel_filter Argument	Action
Zero or not specified	No filtering of any messages. All messages can be retrieved.
Source **`q_address`**	Only those messages that have a matching **`q_address`** are retrieved.
Selection mask created with `pams_set_select`	Only messages that exactly match the specified selection mask are retrieved.

Compound Selection

Allows the application to formulate complex rules for the order in which the message queues are searched. The `pams_set_select` function allows the application to create custom selection masks that can be used in the low-order word of the **`sel_filter`** argument. The format for compound selection follows:



ZK9034AGE

psb

Receives a PAMS Status Block containing the final completion status. The **`psb`** argument is used when sending or receiving recoverable messages. The PSB structure stores the status information from the message recovery system and may be checked after sending or receiving a message. The structure of the PSB is as follows:

Table 8-36

Low Byte	High Byte	Contents	Description
0	1	Type	PSB type
2	3	Call Dependent	Currently not used.
4	7	PSB Delivery Status	The completion status of the function. It contains the status from MRS. It can also contain a value of `PAMS__SUCCESS` when the message is not sent recoverably.
8	15	Message Sequence Number	A unique number assigned to a message when it is sent and follows the message to the destination PSB. This number is input to the `pams_confirm_msg` function to release a recoverable message.
16	19	PSB UMA Status	The completion status of the undeliverable message action (UMA). The PSB UMA status indicates if the UMA was not executed or applicable.
20	23	Function Return Status	After a BEA MessageQ function completes execution, BEA MessageQ software writes the return value to this field.
24	31	Not Used	Not used.

show_buffer

Receives additional information which BEA MessageQ extracts from the message header. The structure of the **`show_buffer`** argument is as follows:

Table 8-37

Longword	Contents	Description
0	Version	The version of the **`show_buffer`** structure. Valid values are as follows: 10 = Version 1.0 20 = Version 2.0
1	Transfer Status	The status code associated with the transfer of **`show_buffer`** information into the application's buffer. Valid symbols are as follows: `PAMS__SUCCESS`—All available information has been transferred. `PAMS__BUFFEROVF`—Information was lost due to receiver buffer overflow. 0—No message returned. There is no information to transfer.
2	Transfer Size	The number of bytes transferred to the application buffer.

3	Flags	A bit array showing the status of fields in the `show_buffer`. A set bit indicates a valid field, while a cleared bit indicates indeterminable data or the end of the allocated `show_buffer` memory. The symbols for the flags field are as follows:
		`PSYM_SHOW_VERSION`
		`PSYM_SHOW_STATUS`
		`PSYM_SHOW_SIZE`
		`PSYM_SHOW_FLAGS`
		`PSYM_SHOW_TARGET`
		`PSYM_SHOW_ORIGINAL_TARGET`
		`PSYM_SHOW_SOURCE`
		`PSYM_SHOW_ORIGINAL_SOURCE`
		`PSYM_SHOW_DELIVERY`
		`PSYM_SHOW_PRIORITY`
		`PSYM_SHOW_ENDIAN`
		`PSYM_SHOW_CORRELATION_ID`

4	Not Used	Fills out the Control Section to its maximum 24 bytes.
5	Not Used	Fills out the Control Section to its maximum 24 bytes.

Table 8-37

Longword	Contents	Description
6	Not Used	Fills out the Control Section to its maximum 24 bytes.
7	Not Used	Fills out the Control Section to its maximum 24 bytes.
8	Not Used	Fills out the Control Section to its maximum 24 bytes.
9	Not Used	Fills out the Control Section to its maximum 24 bytes.
10	Target	The **`q_address`** of the latest message target.
11	Original Target	The **`q_address`** of the original message target.
12	Source	The **`q_address`** of the latest message source.
13	Original Source	The **`q_address`** of the original message.
14	Delivery Mode	The delivery mode that was used to queue the message.
15	Priority	The priority used to queue the message.

16	Endian	The byte ordering or encoding schemes of 2- and 4-byte integers. The possible settings are as follows:
		`PSYM_UNKNOWN`
		`PSYM_VAX_BYTE_ORDER` or `PSYM_LITTLE_ENDIAN`
		`PSYM_NETWORK_BYTE_ORDER` or
		`PSYM_BIG_ENDIAN`
		`PSYM_FML`

17	Correlation ID	The 32 byte correlation ID associated with the message.

show_buff_len

Supplies the length in bytes of the buffer defined in the **`show_buffer`** argument. The minimum length is 40 bytes. If the buffer is too small to contain all of the information, the return code `PAMS__BUFFEROVF` will be in the **`show_buffer`** transfer status.

large_area_len

Specifies the size of the message area to receive messages larger than 32K. Also specifies the length of the message buffer when using double buffers (as indicated by `PSYM_MSG_BUFFER_POINTER`). This argument also stores the length of double buffers and FML32 buffers after reallocation.

large_size

Returns the actual size of the large message, double buffer message, or FML message written to the message buffer.

nullarg_3

Reserved for BEA MessageQ internal use as a placeholder argument. This argument must be supplied as a null pointer.

Return Codes

Table 8-38

Return Code	Platform	Description
`PAMS__AREATOSMALL`	All	Received message is larger than the application message area.
`PAMS__BADARGLIST`	All	Wrong number of call arguments have been passed to this function.
`PAMS__BADHANDLE`	All	Invalid message handle.
`PAMS__BADPARAM`	All	Bad argument value.
`PAMS__BADPRIORITY`	All	Invalid priority value used for receive.
`PAMS__BADSELIDX`	All	Invalid or undefined selective receive index.
`PAMS__BADTIME`	OpenVMS	An invalid time was specified.
`PAMS__BUFFEROVF`	UNIX Windows NT	The size specified for the `show_buffer` argument is too small.
`PAMS__EXHAUSTBLKS`	OpenVMS	No more message blocks available.
`PAMS__FMLERROR`	All	Problem detected with internal format of FML message; this can be an error in processing or data corruption.
`PAMS__INSQUEFAIL`	All	Failed to properly queue a message buffer.
`PAMS__MSGTOSMALL`	All	The **`msg_area_len`** argument must be positive or zero.
`PAMS__MSGUNDEL`	All	Message returned is undeliverable.

Table 8-38

Return Code	Platform	Description
PAMS__NEED_BUFFER_PTR	UNIX Windows NT	FML32 buffer received but **msg_area_len** argument not set to PSYM_MSG_BUFFER_PTR.
PAMS__NETERROR	Clients	Network error resulted in a communications link abort.
PAMS__NOACCESS	All	No access to resource. ACL check failed.
PAMS__NOACL	All	The queue access control file could not be found.
PAMS__NOMEMORY	OpenVMS	Insuffucient memory resources to reallocate buffer pointer.
PAMS__NOMRQRESRC	All	Insufficient multireader queue resources to allow access.
PAMS__NOTDCL	All	Process has not been attached to BEA MessageQ.
PAMS__NOTSUPPORTED	UNIX Windows NT	Specified delivery mode is not supported.
PAMS__PAMSDOWN	UNIX Windows NT	The specified BEA MessageQ group is not running.
PAMS__PREVCALLBUSY	Clients	Previous call to CLS has not been completed.
PAMS__QUECORRUPT	OpenVMS	Message buffer queue corrupt.
PAMS__REMQUEFAIL	All	Failed to properly read a message buffer.
PAMS__STALE	All	Resource is no longer valid and must be freed by the user.
PAMS__STOPPED	All	The requested queue has been stopped.
PAMS__SUCCESS	All	Successful completion.
PAMS__TIMEOUT	All	Timeout period has expired.
PAMS__CONFIRMREQ	All	Confirmation required for this message.

Table 8-38

Return Code	Platform	Description
`PAMS__PAMSDOWN`	UNIX Windows NT	The specified BEA MessageQ group is not running.
`PAMS__POSSDUPL`	All	Message is a possible duplicate.
`PAMS__SUCCESS`	All	Indicates successful completion.

See Also

- `pams_get_msga`
- `pams_put_msg`
- `pams_set_select`

Example

Block Until a Message Is Read

This example shows how to use the `pams_get_msgw` function. It sets an alarm to send messages to itself every 5 seconds; it uses `pams_get_msgw` to sit and wait for them. The queue named “`queue_1`” must be defined in your initialization file as a primary queue. The complete code example called `x_getw.c` is contained in the examples directory.

pams_locate_q

Locates the queue address for the specified queue name or queue alias. By default, this function waits for the queue address to be returned.

Syntax

```
int32 pams_locate_q ( q_name, q_name_len, q_address, [wait_mode],
              [req_id], [resp_q], [name_space_list],
              [name_space_list_len], [timeout] )
```

Arguments

Table 8-39

Argument	Data Type	Mechanism	Prototype	Access
q_name	char	reference	char *	passed
q_name_len	int32	reference	int32 *	passed
q_address	q_address	reference	q_address *	returned
[wait_mode]	int32	reference	int32 *	passed
[req_id]	int32	reference	int32 *	passed
[resp_q]	int32	reference	int32 *	passed
[name_space_list]	int32 array	reference	int32 array *	passed
[name_space_list_len]	int32	reference	int32 *	passed
[timeout]	int32	reference	int32 *	passed

Argument Definitions

q_name

Supplies the queue name or queue alias whose queue address is requested. The procedure that BEA MessageQ uses to find this name is controlled by the **`name_space_list`** argument, described below.

q_name_len

Supplies the number of characters in the **`q_name`** argument. The maximum string length on UNIX, Windows NT, and OpenVMS systems is 255 characters. For all other BEA MessageQ environments, the maximum string length is 31.

q_address

Receives the queue address assigned by BEA MessageQ when an application has successfully located the queue name.

wait_mode

Supplies the search mode of the `pams_locate_q` function. The mode indicates whether the application waits for the search completion or receives the response in an acknowledgment message. There are two predefined constants for this argument:

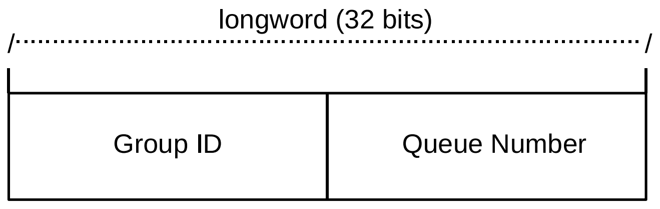
- `PSYM_WF_RESP` (default setting)—The application issues the `pams_locate_q` request and waits for the queue address to be returned.
- `PSYM_AK_RESP`—The application issues the `pams_locate_q` address and continues processing. When the search is completed, the queue address is returned to the application's primary queue in a `LOCATE_Q_REP` message. The response message can be redirected to an alternate queue address using the **`resp_q`** argument.

req_id

Supplies an application-specified transaction ID to associate with the `pams_locate_q` function.

resp_q

Supplies an alternate queue to use for receiving the acknowledgment message of the **`q_address`**. If no response queue is specified, the acknowledgment message is sent to the sender program's primary queue. The **`resp_q`** argument has the following format:



ZK9007AGE

Note that the group ID field is always equal to zero because the sender program cannot specify a response queue outside its group.

name_space_list

If the **`name_space_list`** argument is specified, the **`name_space_list_len`** argument must also be specified. If this argument is unspecified, then `PSEL_TBL_GRP` is the default.

Possible values in a **`name_space_list`** argument are as follows:

Location it represents	Symbolic value
Process cache	`PSEL_TBL_PROC`
Group/group cache	`PSEL_TBL_GRP`
Global name space	`PSEL_TBL_BUS` (or `PSEL_TBL_BUS_MEDIUM` or `PSEL_TBL_BUS_LOW`)

The **`name_space_list`** argument identifies the scope of the name as follows:

- To identify a local queue reference or a queue, an application must include `PSEL_TBL_GRP` in **`name_space_list.`** (Do not specify `PSEL_TBL_BUS` in the list because it would identify a global queue reference.)
- To identify a global queue reference, include `PSEL_TBL_BUS` (or `PSEL_TBL_BUS_MEDIUM` or `PSEL_TBL_BUS_LOW`) in the **`name_space_list`** argument and specify its pathname, either explicitly or implicitly. If the **`q_name`** argument contains any slashes (`/`), or periods (`.`), BEA MessageQ treats it as a pathname. Otherwise, BEA MessageQ treats **`q_name`** as a name and adds the `DEFAULT_NAMESPACE_PATH` to the name to create the pathname to lookup. (The `DEFAULT_NAMESPACE_PATH` is set in the `%PROFILE` section of the group initialization file.)

The **`name_space_list`** argument also controls the cache access as follows:

- To lookup a local queue reference or queue name, specify both `PSEL_TBL_GRP` and `PSEL_TBL_PROC`. This causes the process cache to be checked before looking into the group cache.
- To lookup a global queue reference, specify `PSEL_TBL_BUS` (or `PSEL_TBL_BUS_LOW` or `PSEL_TBL_BUS_MEDIUM`), `PSEL_TBL_GRP` and `PSEL_TBL_PROC`. This causes the process cache to be checked. Then, the group cache is checked before looking into the global name space.

Note that to lookup all caches in the global name space before looking in the master database, specify `PSEL_TBL_BUS_LOW` instead of `PSEL_TBL_BUS`.

To lookup only the slower but more up-to-date caches in the global name space before looking in the master database, specify `PSEL_TBL_BUS_MEDIUM` instead of `PSEL_TBL_BUS`.

For more information on dynamic binding of queue addresses, see the Using Naming topic.

name_space_list_len

Supplies the number of entries in the **`name_space_list`** argument. If the **`name_space_list_len`** argument is zero, BEA MessageQ uses `PSEL_TBL_GRP` as the default in the **`name_space_list`** argument.

timeout

Specifies the number of PAMS time units (1/10 second intervals) to allow for the locate to complete. If timeout is zero, the group's `ATTACH_TMO` property is used. If the `ATTACH_TMO` is also zero, 600 is used.

Return Values

Table 8-40

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Wrong number of call arguments.
`PAMS__BADNAME`	UNIX Windows NT	The queue name contains illegal characters.
`PAMS__BADPARAM`	All	Invalid argument in the argument list.
`PAMS__BADRESPQ`	All	Invalid response queue specified.
`PAMS__BOUND`	All	Queue name in use.
`PAMS__BUSNOTSET`	UNIX Windows NT	`DMQ_BUS_ID` environment variable not set.
`PAMS__GROUPNOTSET`	UNIX Windows NT	`DMQ_GROUP_ID` environment variable not set.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.

Table 8-40

Return Code	Platform	Description
`PAMS__NOACCESS`	All	The address of the specified name is either 0 or is in another group.
`PAMS__NOOBJECT`	All	Could not locate queue name.
`PAMS__PAMSDOWN`	All	The specified BEA MessageQ group is not running.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__RESRCFAIL`	All	Failed to allocate resources.
`PAMS__SUCCESS`	All	Successful completion of an action.
`PAMS__TIMEOUT`	All	The timeout period specified has expired.
`PAMS__UNBINDING`	All	Queue requested is in the process of unbinding from a `pams_bind_q` request.

See Also

- `pams_attach_q`
- `pams_exit`

Example

Locate a Queue Address

This example shows how to use the `pams_locate_q` function by attaching to `queue_1` and locating `queue_3` where a message is being sent. The queues named “`queue_1`” and “`queue_3`” must be defined in your initialization file; `queue_1` must be a primary queue. The complete code example called `x_locate.c` is contained in the examples directory.

pams_open_jrn

Opens the selected message recovery journal. The BEA MessageQ dead letter journal (DLJ) stores messages designated as recoverable that could not be delivered by the recovery system. The BEA MessageQ postconfirmation journal (PCJ) stores recoverable messages that were successfully delivered. See the Using Recoverable Messaging topic for a description of BEA MessageQ message recovery services.

Syntax

```
int32 pams_open_jrn ( jrn_filespec, jrn_filename_len, jrn_handle )
```

Arguments

Table 8-41

Argument	Data Type	Mechanism	Prototype	Access
`jrn_filespec`	`char`	reference	`char *`	passed
`jrn_filename_len`	`short`	reference	`short *`	passed
`jrn_handle`	`int32`	reference	`int32 *`	returned

Argument Definitions

jrn_filespec

Supplies the file name of the message recovery journal from which the application will read stored messages.

jrn_filename_len

Supplies the length of the file specification entered to the **`jrn_filespec`** argument specified (filename array) in number of bytes.

jrn_handle

Receives the journal handle for the selected message recovery file if this function completes successfully.

Return Values

Table 8-42

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Invalid number of call arguments.
`PAMS__NOMEMORY`	OpenVMS	Insufficient virtual memory.

Table 8-42

Return Code	Platform	Description
`PAMS__NOSUCHPCJ`	OpenVMS	Error occured when attempting to open the specified journal file.
`PAMS__SUCCESS`	OpenVMS	Indicates successful completion.

See Also

- `pams_close_jrn`
- `pams_confirm_msg`
- `pams_put_msg`
- `pams_read_jrn`

pams_put_msg

Sends a message to a target queue using a set of standard BEA MessageQ delivery modes. Applications specify buffer-style or FML-style messaging using the **msg_size** argument. For buffer-style messaging using message buffers up to 32K, this argument supplies the length of the message in bytes in the user's **msg_area** buffer. In addition, you can use the **msg_size** argument to specify one of the following symbols:

- PSYM_MSG_FML—indicates FML-style messaging. The **msg_area** argument must contain a pointer to an FML32 buffer.
- PSYM_MSG_LARGE—indicates buffer-style message with messages up to 4MB in length. The pointer to the buffer is contained in the **msg_area** argument and the size of the large message buffer is contained in the **large_size** argument.

The **delivery** argument of the pams_put_msg function can be used to guarantee message delivery if a system, process, or network fails. Recoverable messages are stored on disk by the message recovery system until they can be delivered to the target queue of the receiver program. When sending a recoverable message, you must specify the **uma** argument if the message recovery cannot store the message. You must also supply the **psb** argument to receive the return status of the operation.

The optional **timeout** argument lets you set a maximum amount of time for the send operation to complete before the function times out. The optional **resp_q** argument allows you to specify an alternate queue for receiving the response messages rather than directing responses to the primary queue of the sender program.

To use a pointer to an FML32 buffer when sending a message, the sender program specifies the symbol PSYM_MSG_FML as the **msg_size** argument to the pams_put_msg function.

Syntax

```
int32 pams_put_msg ( msg_area, priority, target, class, type,
              delivery, msg_size, [timeout], [psb], [uma],
              [resp_q], [large_size], [correlation_id],
              [nullarg_3] )
```

Arguments

Table 8-43

Argument	Data Type	Mechanism	Prototype	Access
msg_area	char	reference	char *	passed
priority	char	reference	char *	passed

Table 8-43

Argument	Data Type	Mechanism	Prototype	Access
`target`	`q_address`	reference	`q_address *`	passed
`class`	`short`	reference	`short *`	passed
`type`	`short`	reference	`short *`	passed
`delivery`	`char`	reference	`char *`	passed
`msg_size`	`short`	reference	`short *`	passed
`[timeout]`	`int32`	reference	`int32 *`	passed
`[psb]`	`struct psb`	reference	`struct psb *`	returned
`[uma]`	`char`	reference	`char *`	passed
`[resp_q]`	`q_address`	reference	`q_address *`	passed
`large_size`	`int32`	reference	`int32 *`	passed
`[correlation_id]`	`char`	reference	`char *`	passed
`[nullarg_3]`	`char`	reference	`char *`	passed

Argument Definitions

msg_area

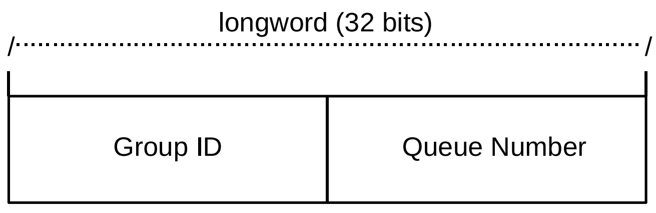
For buffer-style messaging, supplies the address of a memory region or a message pointer containing the message to be delivered to the target queue of the receiver program. For FML-style messaging, supplies the message pointer that points to an FML32 buffer containing the message.

priority

Supplies the priority level for selective message reception. Priority ranges from 0 (lowest priority) to 99 (highest priority).

target

Supplies the queue number and group ID of the receiver program’s queue address in the following format:



ZK9007AGE

class

Supplies the class code of message being sent. BEA MessageQ supports the use of symbolic names for **`class`** argument values. Symbolic class names should begin with `MSG_CLAS_`. For information on defining class symbols, see the `p_typecl.h` include file. On UNIX and Windows NT systems, the `p_typecl.h` include file cannot be edited. You must create an include file to define type and class symbols for use by your application.

Class symbols reserved by BEA MessageQ are as follows:

Reserved Class	**Symbol Value**
`MSG_CLAS_MRS`	28
`MSG_CLAS_PAMS`	29
`MSG_CLAS_ETHERNET`	100
`MSG_CLAS_UCB`	102
`MSG_CLAS_TUXEDO`	31001
`MSG_CLAS_TUXEDO_TPSUCCESS`	31002
`MSG_CLAS_TUXEDO_TPFAIL`	31003
`MSG_CLAS_XXX`	30000 through 32767 (except 31001-31003)

type

Supplies the type code for the message being sent. BEA MessageQ supports the use of symbolic names for **`type`** argument values. Symbolic type names begin with `MSG_TYPE_`. For information on defining type symbols, see the `p_typecl.h` include file.

BEA MessageQ has reserved the symbol value range -1 through -5000. A zero value for this argument indicates that no processing by message type is expected.

delivery

Supplies the delivery mode for the message using the following format:

- `PDEL_MODE_`*`sn_dip`*—where *`sn`* is one of the following sender notification constants:
- `WF`—Wait for completion
- `AK`—Asynchronous acknowledgment
- `NN`—No notification

And *`dip`* is one of the following delivery interest point constants:

- `ACK`—Read from target queue and explicitly acknowledged using the `pams_confirm_msg` function. ACK can also be an implicit acknowledgement sent after the second `pams_get_msg` call by the receiving application.
- `CONF`—Delivered from the DQF and explicitly confirmed using the `pams_confirm_msg` function (recoverable)
- `DEQ`—Read from the target queue
- `DQF`—Stored in the destination queue file (recoverable)
- `MEM`—Stored in the target queue
- `SAF`—Stored in the store and forward file (recoverable)

Note: If temporary queues are used, deleted, and reused quickly, it is possible in isolated cases for an implicit `ACK` response from a previous temporary queue to be placed on the new temporary queue.

msg_size

For buffer-style messaging using message buffers up to 32K, supplies the length of the message in bytes in the user's **`msg_area`** buffer. In addition, you can specify one of the following symbols:

- `PSYM_MSG_FML`—-indicates FML-style messaging. The **`msg_area`** argument must contain a pointer to an FML32 buffer.
- `PSYM_MSG_LARGE`—indicates buffer-style messaging with messages up to 4MB in length. The pointer to the buffer is contained in the **`msg_area`** argument and the size of the large message buffer is contained in the **`large_size`** argument.

timeout

Supplies the maximum amount of time the `pams_put_msg` function waits for a message to arrive before returning control to the application. The timeout value is entered in tenths (0.1) of a second. A value of 100 indicates a timeout of 10 seconds. If the timeout occurs before a message arrives, the status code `PAMS__TIMEOUT` is returned. Specifying 0 as the timeout value sets the timeout to the default value of 30 seconds.

psb

Receives a value in the PAMS Status Block specifying the final completion status. The **`psb`** argument is used when sending or receiving recoverable messages. The PSB structure stores the status information from the message recovery system and may be checked after sending or receiving a message.

The structure of the PSB is as follows:

Table 8-44

Low Byte	High Byte	Contents	Description
1	0	Type	PSB type.
3	2	Call Dependent	Currently not used.
7	4	PSB Delivery Status	The completion status of the function. It contains the status from MRS. It can also contain a value of `PAMS__SUCCESS` when the message is not sent recoverably.

Table 8-44

Low Byte	High Byte	Contents	Description
15	8	Message Sequence Number	A unique number assigned to the message when it is sent and follows the message to the destination PSB. This number is input to the `pams_confirm_msg` function to release a recoverable message.
19	16	PSB UMA Status	The completion status of the undeliverable message action (UMA). The PSB UMA status indicates if the UMA was not executed or applicable.
23	20	Function Return Status	After a BEA MessageQ function completes execution, BEA MessageQ software writes the return value to this field.
31	24	Not Used	Not used.

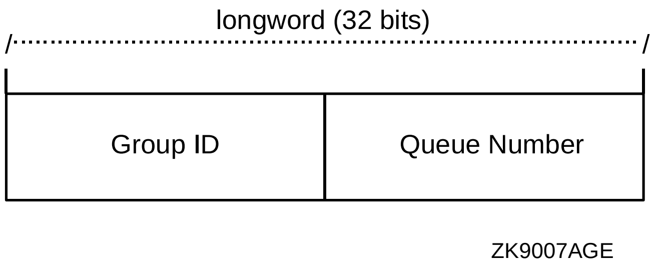
uma

Supplies the action to be performed if the message cannot be stored at the specified delivery interest point. The format of this argument is `PDEL_UMA_`*`XXX`* where *XXX* is one of the following symbols:

Symbol	Description
`DISC`	Discard message
`DISCL`	Discard after logging message
`DLJ`	Dead letter journal
`DLQ`	Dead letter queue
`RTS`	Return to sender
`SAF`	Store and Forward

resp_q

Supplies a **`q_address`** to use as the alternate queue for receiving response messages from the receiver program. The sender program must be attached to the queue specified in the **`resp_q`** argument to receive the response messages. The **`resp_q`** argument has the following format:



The group ID is always specified as zero because the sender program cannot assign a response queue outside its group.

large_size

Supplies the actual size of the large message written to the message buffer.

correlation_id

Supplies the correlation id, a user-defined identifier stored as a 32-byte value

nullarg_3

Reserved for BEA MessageQ internal use as a placeholder argument. This argument must be supplied as a null pointer.

Return Values

Table 8-45

Return Code	Platform	Description
`PAMS__BADARGLIST`	All	Wrong number of call arguments have been passed to this function.
`PAMS__BADDELIVERY`	All	Invalid delivery mode.
`PAMS__BADHANDLE`	All	Invalid message handle.

Table 8-45

Return Code	Platform	Description
`PAMS__BADPARAM`	UNIX Windows NT OpenVMS	Attempt to use cross-group connection when cross-group communication is disabled. On OpenVMS systems, invalid NULL call argument.
`PAMS__BADPRIORITY`	All	Invalid priority value on send operation.
`PAMS__BADPROCNUM`	UNIX Windows NT	Invalid target queue address specified.
`PAMS__BADRESPQ`	All	Response queue not owned by process.
`PAMS__BADTIME`	OpenVMS	Invalid time specified.
`PAMS__BADUMA`	All	Undeliverable message action (UMA) is invalid.
`PAMS__EXCEEDQUOTA`	All	Target process quota exceeded; message was not sent.
`PAMS__EXHAUSTBLKS`	OpenVMS	No more message blocks available.
`PAMS__FMLERROR`	All	Problem detected with internal format of FML message; this can be an error in processing or data corruption.
`PAMS__LINK_UP`	OpenVMS	MRS has reestablished link.
`PAMS__MSGTOBIG`	All	Message exceeded the size of the largest link list section (LLS).
`PAMS__MSGTOSMALL`	OpenVMS	Invalid (negative) `msg_size` specified in the argument list.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOMRS`	OpenVMS	MRS is not available.
`PAMS__NOTACTIVE`	All	Target process is not currently active; message not sent.

Table 8-45

Return Code	**Platform**	**Description**
`PAMS__NOTDCL`	All	Process has not been attached to BEA MessageQ.
`PAMS__NOTFLD`	All	The buffer supplied is not an FML32 buffer.
`PAMS__NOTSUPPORTED`	All	The combination of delivery mode and uma selected is not supported.
`PAMS__PNUMNOEXIST`	OpenVMS	Target queue number does not exist.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__REMQUEFAIL`	All	Failed to properly dequeue a message buffer.
`PAMS__STOPPED`	All	The requested queue has been stopped.
`PAMS__SUCCESS`	All	Successful completion.
`PAMS__TIMEOUT`	All	Timeout period has expired.
`PAMS__UNATTACHEDQ`	All	Message successfully sent to unattached queue.
`PAMS__WAKEFAIL`	OpenVMS	Failed to wake up the target process.

Table 8-46

UMA Status	Platform	Description
`PAMS__DISC_FAILED`	All	Message not recoverable in destination queue file (DQF); undeliverable message action (UMA) was `PDEL_UMA_DISC`; message could not be discarded.
`PAMS__DISC_SUCCESS`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_DISC`; message discarded.

Table 8-46

UMA Status	Platform	Description
`PAMS__DISCL_FAILED`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_DISC`; recoverability failure could not be logged or message could not be discarded.
`PAMS__DISCL_SUCCESS`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_DISC`; message discarded after logging recoverability failure.
`PAMS__DLJ_FAILED`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_DLJ`; dead letter journal (DLJ) write operation failed.
`PAMS__DLJ_SUCCESS`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_DLJ`; message written to the DLJ.
`PAMS__DLQ_FAILED`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_DLQ`; message could not be queued to the DLQ.
`PAMS__DLQ_SUCCESS`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_DLQ`; message queued to the DLQ.
`PAMS__NO_UMA`	All	Message is recoverable; UMA not executed.
`PAMS__RTS_FAILED`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_RTS`; message could not be returned to sender.
`PAMS__RTS_SUCCESS`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_RTS`; message returned to sender.
`PAMS__SAF_FAILED`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_SAF`; store and forward (SAF) write operation failed.
`PAMS__SAF_SUCCESS`	All	Message not recoverable in DQF; UMA was `PDEL_UMA_SAF`; message recoverable from SAF file.

Table 8-46

UMA Status	Platform	Description
`PAMS__UMA_NA`	All	UMA not applicable.

See Also

- `pams_get_msg`
- `pams_get_msga`
- `pams_get_msgw`

Example

Send a Message

This example sends a number of messages to a queue. The complete code example called `x_putslf.c` is contained in the examples directory.

pams_read_jrn

Reads a message from a BEA MessageQ journal file. Use the `pams_open_jrn` function to open the dead letter journal or postconfirmation journal for a message queuing group. Use the `pams_close_jrn` function to close the journal file after reading selected messages. Note that on UNIX and Windows NT systems, these functions are performed by running the Journal Replay utility.

The receiver program determines whether each message is a FML buffer or a large message by reading the **`len_data`** argument. See the Sending and Receiving BEA MessageQ Messages topic for more information on working with message handles and large messages.

Syntax

```
int32 pams_read_jrn ( jrn_handle, msg_area, priority, source,
              class, type, msg_area_len, len_data, target,
              write_time, conf_val, msg_seq_num, mrs_status,
              [large_area_len], [large_size], [nullarg_3] )
```

Arguments

Table 8-47

Argument	Data Type	Mechanism	Prototype	Access
`jrn_handle`	`int32`	reference	`int32 *`	passed
`msg_area`	`char`	reference	`char *`	returned
`priority`	`char`	reference	`char *`	returned
`source`	`q_address`	reference	`q_address *`	returned
`class`	`short`	reference	`short *`	returned
`type`	`short`	reference	`short *`	returned
`msg_area_len`	`short`	reference	`short *`	returned
`len_data`	`short`	reference	`short*`	returned
`target`	`q_address`	reference	`q_address *`	returned
`write_time`	`unsigned int32`	reference	`unsigned int32 *`	returned
`conf_val`	`int32`	reference	`int32 *`	returned

Table 8-47

Argument	Data Type	Mechanism	Prototype	Access
msg_seq_num	unsigned int32	reference	unsigned int32 *	returned
mrs_status	int32	reference	int32 *	returned
[large_area_len]	int32	reference	int32 *	returned
[large_size]	int32	reference	int32 *	returned
[nullarg_3]	char	reference	char *	returned

Argument Definitions

jrn_handle

Supplies the journal handle of the message recovery journal from which the application has selected to read journal entries. The journal handle is returned to the application by the `pams_open_jrn` function.

msg_area

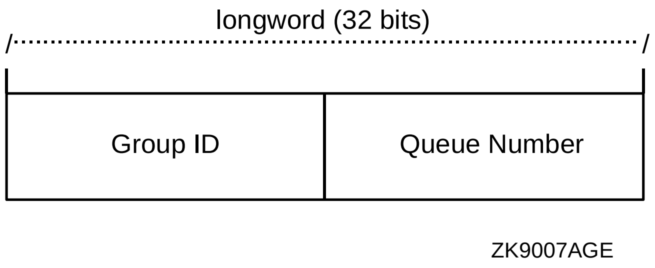
Receives the contents of the message retrieved from the selected message recovery journal. This argument contains either the address of a memory region or a message handle where BEA MessageQ writes.

priority

Supplies the priority level for selective message reception. Priority ranges from 0 (lowest priority) to 99 (highest priority).

source

Receives a structure containing the queue number and group ID of the sender program's primary queue in the following format:



class

Receives the class code of the retrieved message. The class is specified in the arguments of the `pams_put_msg` function. BEA MessageQ supports the use of symbolic names for **`class`** argument values. Symbolic class names should begin with `MSG_CLAS_`. For information on defining class symbols, see the `p_typecl.h` include file. Class symbols reserved by BEA MessageQ are as follows:

Reserved Class	Symbol Value
`MSG_CLAS_MRS`	28
`MSG_CLAS_PAMS`	29
`MSG_CLAS_ETHERNET`	100
`MSG_CLAS_UCB`	102
`MSG_CLAS_TUXEDO`	31001
`MSG_CLAS_TUXEDO_TPSUCCESS`	31002
`MSG_CLAS_TUXEDO_TPFAIL`	31003
`MSG_CLAS_XXX`	30000 through 32767 (except 31001-31003)

type

Receives the type code of the journaled message. The type is specified in the arguments of the `pams_put_msg` function. BEA MessageQ supports the use of symbolic names for **`type`** argument values. Symbolic type names begin with `MSG_TYPE_`. For information on defining type symbols, see the `p_typecl.h` include file. The OpenVMS symbol values range from –1 through –5000. Use of the **`type`** argument facilitates selective message reception. However, if the receiving application does not need a specific value for its processing, then use a value of 0.

msg_area_len

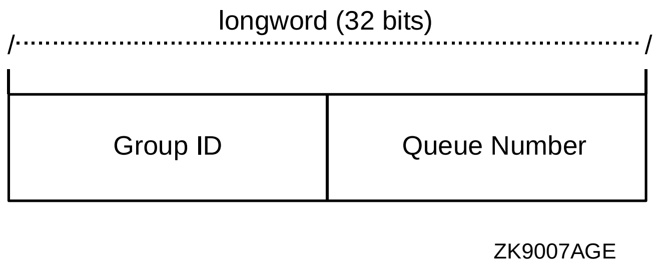
Supplies the size of the buffer (in bytes) for buffer-style messages of up to 32K bytes. The **`msg_area`** buffer is used to store the retrieved message.

len_data

- For buffer-style messaging with messages of up to 32K, this argument receives the number of bytes retrieved from the message queue and stored in the area specified by the **`msg_area`** argument.
- For an FML-style message, this argument contains the symbol `PSYM_MSG_BUFFER_PTR` to indicate that the message is a pointer to an FML32 buffer.
- For large messages, this argument contains the symbol `PSYM_MSG_LARGE` to indicate that the message buffer is greater than 32K. The size of the message is returned in the **`large_size`** argument.

target

Receives the queue number and group ID of the receiver's queue address in the following format:



write_time

Receives the address of the quadword (an array of two int32 values) specifying the date and time that the recoverable message was confirmed. This parameter uses standard OpenVMS system time.

conf_val

Receives the message confirmation value.

msg_seq_num

Receives the message sequence number generated by BEA MessageQ in the PSB of the received message. This argument should be set to the values in the PSB.

mrs_status

Receives the Message Recovery Services (MRS) status of the message.

large_area_len

Specifies the size of the message buffer to receive messages larger than 32K.

large_size

Returns the actual size of the large message written to the message buffer.

nullarg_3

Reserved for BEA MessageQ internal use as a placeholder argument. This argument must be supplied as a null pointer.

Return Values

Table 8-48

Return Code	Platform	Description
`PAMS__AREASTOSMALL`	OpenVMS	Received message is larger than the user message area.
`PAMS__BADARGLIST`	OpenVMS	Invalid number of arguments supplied.
`PAMS__BADHANDLE`	OpenVMS	Invalid message handle.
`PAMS__INVJH`	OpenVMS	Invalid journal handle.
`PAMS__MSGTOBIG`	OpenVMS	Message in journal file is larger than `GROUP_MAX_MESSAGE_SIZE`.
`PAMS__NOMEMORY`	OpenVMS	Insufficient virtual memory.
`PAMS__NOMOREMSG`	OpenVMS	No more messages in journal.

Table 8-48

Return Code	Platform	Description
`PAMS__SUCCESS`	OpenVMS	Indicates successful completion.

See Also

- `pams_close_jrn`
- `pams_open_jrn`

pams_set_select

Allows application developers to define complex selection criteria for message reception. The selection array specifies the queues to search, the priority order of message reception, two comparison keys for range checking, and an order key to determine the order in which messages are selected from the queue.

The `pams_set_select` function creates an index handle that is used as the **`sel_filter`** argument of BEA MessageQ functions for reading the message. When a selection index handle is passed to `pams_get_msg`, `pams_get_msga` or `pams_get_msgw`, each message received is compared against comparison `key_1` and then comparison `key_2`. If the message matches both keys (a logical AND operation), the message is added to a set of matched messages. The order in which selected messages are delivered is determined by the order key.

Syntax

```
int32 pams_set_select ( selection_array, num_masks, index_handle )
```

Arguments

Table 8-49

Argument	Data Type	Mechanism	Prototype	Access
`selection_array`	`selection_array_component`	reference	`selection_array_component *`	passed
`num_masks`	`short`	reference	`short *`	passed
`index_handle`	`int32`	reference	`int32 *`	returned

Argument Definitions

selection_array

Supplies an array of selection records that contain the selection rules for each queue. The `typedef` structures define the C data structure for the selection array. The structure is defined in `p_entry.h` as follows:

```
typedef struct _selection_array_component {
    int32 queue;
    int32 priority;
    int32 key_1_offset;
    int32 key_1_size;
    int32 key_1_value;
    int32 key_1_oper;
    int32 key_2_offset;
    int32 key_2_size;
    int32 key_2_value;
    int32 key_2_oper;
    int32 order_offset;
    int32 order_size;
    int32 order_order;
    union {
       pams_correlation_id  correlation_id;
       pams_sequence_number sequence number
    } extended_key
} selection_array_component;
```

The `selection_array_component` data structure has the following components:

Component	Description
Queue and Priority	Allows the application to specify the queue number and priority.
Comparison Key 1	Defines the components of the first comparison key used to enable range checking of messages.
Comparison Key 2	Defines the components of the second comparison key used to enable range checking of messages.
Order Key	Contains the information required to provide selection of messages by FIFO, Minimum Value, or Maximum Value.

The following tables define the content of each of the components of the `selection_array_component` data structure.

Queue and Priority

The following table specifies the valid values that can be applied to the arguments in this part of the `Select_Queue` structure:

Table 8-50

Field	Values	Description
Queue	Queue Number	Specifies the queue number to be searched. The queue number can be any message queue for which the application has read access. The queue number can be obtained from the **`q_attached`** argument of the `pams_attach_q` function or **`q_address`** of the `pams_locate_q` function. A value of 0 for this argument specifies the application's primary queue.
Priority		Specifies the priority, using either an integer between 0 and 99 inclusive or a variable. (Using the direct interger value is the preferred method of specifying priority.) This argument also accepts the following predefined constants which are set by the application.
	`PSEL_PRI_ANY`	Read priority 1 before reading priority 0 messages.
	`PSEL_PRI_P0`	Read priority 0 messages only.
	`PSEL_PRI_P1`	Read priority 1 messages only.

Comparison Keys

The following table specifies the arguments and valid values that can be applied to this part of the `Selection_Array_Components` structure:

Table 8-51

Field	Values	Description
Offset		Contains a value that specifies where the information to be compared begins inside the message. The following predefined constants apply:
	n	User message byte number (0 relative).
	`PSEL_SOURCE`	Source address of message.

Table 8-51

Field	Values	Description
	`PSEL_CLASS`	Class of the message.
	`PSEL_TYPE`	Type of the message.
	`PSEL_CORRELATION_ID`	Correlation ID of the message. May be used for `key_1_offset` or `key_2_offset` but not both. If this symbol is specified, the Size field must be set to `PSEL_CORRELATION_ID_SIZE` (or 32 bytes).
	`PSEL_SEQUENCE_NUMBER`	Message sequence number acquired from the PAMS Status Buffer. If this symbol is specified, the Size field must be set to `PSEL_SEQUENCE_NUMBER_SIZE` (or 8 bytes).
Size		Specifies data type of the key to be compared.
	0	Disable use of key.
	1	Byte (8 bits).
	2	Word (16 bits).
	4	int32 (32 bits).
	`PSEL_SEQUENCE_NUMBER_SIZE`	8 bytes
	`PSEL_CORRELATION_ID_SIZE`	32 bytes
Value	n	Contains the value for message field comparison field that is formatted as an integer of 32 bits.
oper		Relational operator comparison.
	`PSEL_OPER_EQ`	Message field = value.
	`PSEL_OPER_NEQ`	Message field <> value.
	`PSEL_OPER_GTR`	Message field > value.
	`PSEL_OPER_LT`	Message field < value.
	`PSEL_OPER_GTRE`	Message field > or = value.
	`PSEL_OPER_LTE`	Message field < or = value.

Order Key

The Order Key part contains variables described in the following table:

Table 8-52

Field	Values	Description
Offset		Byte offset of the message field. The *offset* variable contains a value that specifies where the information to be compared begins inside the message.
	`n`	User message byte number (0 relative).
	`PSEL_SOURCE`	Source address of the message.
	`PSEL_CLASS`	Class of the message.
	`PSEL_TYPE`	Type of the message.
	`PSEL_CORRELATION_ID`	Correlation ID of the message. If this symbol is specified, the Size field must be set to `PSEL_CORRELATION_ID_SIZE` (or 32 bytes).
	`PSEL_SEQUENCE_NUMBER`	Message sequence number acquired from the PAMS Status Buffer. If this symbol is specified, the Size field must be set to `PSEL_SEQUENCE_NUMBER_SIZE` (or 8 bytes).
Size		Size of the comparison. The *`size`* variable specifies the data type of the key to be compared.
	0	Disable use of key.
	1	Byte.
	2	Word.
	4	`int32` (32 bits).
	`PSEL_SEQUENCE_NUMBER_SIZE`	8 bytes
	`PSEL_CORRELATION_ID_SIZE`	32 bytes
Order		Order operator. The *order* variable specifies the sequence in which the select process is to be performed.

Table 8-52

Field	Values	Description
	PSEL_ORDER_FIFO	First pending.
	PSEL_ORDER_MIN	Minimum value of all pending.
	PSEL_ORDER_MAX	Maximum value of all pending.

Correlation ID

The correlation ID is a 32-byte user-defined identifier associated with a message. If `PSEL_CORRELATION_ID` is supplied as the value for either the `key_1_offset` or key_2_offset field, the correlation ID value is used to match messages with the specified correlation ID. Since there is a single correlation ID per message, `PSEL_CORRELATION_ID` should only be specified for one of the comparison keys; specifying the correlation ID for both keys results in a `PAMS_BADPARAM` error.

If `PSEL_CORRELATION_ID` is supplied as the value for the `order_offset` field, messages with the specified correlation ID are returned in the order specified by the `order_order` field.

Sequence Number

The message sequence number is a unique value for each message. The sequence number is stored in the PAMS Status Buffer (PSB). Applications should acquire the message sequence number from the PSB and not modify it in any way.

Note: An application may specify only one of the two keys to select by correlation identifier or by sequnce number.

num_masks

Supplies the number of records in the selection array. This argument allows a minimum of 1 record to a maximum of 256 records in the selection array.

index_handle

Receives a variable containing the index handle for the selection mask as follows:

- The high-order word contains `PSEL_BY_MASK`.
- The low-order word contains the index to the selection array.

The **`index_handle`** is passed as the **`sel_filter`** argument in `pams_get_msg`, `pams_get_msga` or `pams_get_msgw`, and `pams_cancel_select` functions. OpenVMS allows a maximum number of 500 index handles. Other BEA MessageQ implementations offer 16K to 32K index handles.

Return Values

Table 8-53

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Invalid number of arguments supplied.
`PAMS__BADPARAM`	All	Bad argument passed in the function call.
`PAMS__IDXTBLFULL`	All	Selective receive index table is full.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOTDCL`	All	Process has not been attached to BEA MessageQ.
`PAMS__PAMSDOWN`	UNIX Windows NT	The specified BEA MessageQ group is not running.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__SUCCESS`	All	Indicates successful completion.

See Also

- `pams_cancel_get`
- `pams_cancel_select`
- `pams_get_msg`
- `pams_get_msga`
- `pams_get_msgw`

Example

Selecting Messages Using a Complex Selection Filter

This example shows the selective reception of messages using `pams_set_select` to build a complex message selection filter. The queue named “`queue_1`” must be defined in your initialization file as a primary queue. The complete code example called `x_select.c` is contained in the examples directory.

pams_set_timer

Creates a timer that sends a message to an application's primary queue when a time interval expires or a time of day arrives. The message is sent as a priority 1 message with a source of `PAMS_TIMER_QUEUE`, a class code of PAMS, and a type code of `TIMER_EXPIRED`. A **`timer_id`** is returned by this function as the first int32 value in the `TIMER_EXPIRED` message.

Note: Prior to BEA MessageQ Version 5.0, the valid priority values were 0 and 1. In Version 5.0, the valid range is 0 to 99 (0 being the lowest priority and 99 the highest priority). Keep in mind that timer priorities are always 1 and take this into account when modifying existing programs to take advantage of the expanded priority range. Messages associated with timers have a priority of 1 and are not sent until all messages with priorities from 2 to 99 are read.

To act upon the timer message, the application uses the `pams_get_msgw` function to read its primary queue, block until the timer expiration message arrives, and then act upon it. To cancel a BEA MessageQ timer, use the `pams_cancel_timer` function with the identification code of the timer you want to cancel.

Syntax

```
int32 pams_set_timer ( timer_id, timer_format, p_timeout,
                       s_timeout )
```

Arguments

Table 8-54

Argument	Data Type	Mechanism	Prototype	Access
`timer_id`	`int32`	reference	`int32 *`	passed
`timer_format`	`char`	reference	`char *`	passed
`p_timeout`	`int32`	reference	`int32 *`	passed
`s_timeout`	`unsigned quadword`	reference	`unsigned quadword *`	passed

Argument Definitions

timer_id

Supplies a unique timer identification value created by the application. Must be greater than zero.

timer_format

Supplies the time format being used. Following are the two predefined constants for this argument:

- `P`—selects the time interval in PAMS timer format supplied to the **`p_timeout`** argument. PAMS timer format expresses time in units of one tenth of a second. Using the PAMS timer format provides an operating system independent way to represent a time interval.
- `S`—selects the system-dependent time format supplied to the **`s_timeout`** argument. Using a system-dependent time format limits the portability of applications to a specific operating system environment.

p_timeout

Supplies the amount of time to delay (delta) from the current time before returning a timer expiration message. If the **`timer_format`** argument is set to `P`, a value greater than 0 must be entered for this argument. This argument uses the PAMS timer format which expresses time in units of one tenth of one second.

s_timeout

On OpenVMS systems, use this argument to supply a pointer to an array of two `int32` values used to set a 64-bit OpenVMS time format. The **`s_timeout`** argument can be specified as an absolute time or a delta time matching the OpenVMS time format rules. Note that if the caller exceeds the `ASTLM` or `TQELM` process quota, the process can enter the `RWAST` state.

On UNIX and Windows NT systems, use this argument to supply a two element array of `int32` values. The values represent an absolute time (a UTC time in seconds and microseconds) at which the timer will expire. To use the **`s_timeout`** argument, developers provide a pointer to a "`struct timeval`" as follows:

```
struct timeval theTime;
nStatus = pams_set_timer(&timer_id, “S”, NULL, (int32 *) &theTime);
```

Return Values

Table 8-55

Return Code	Platform	Description
`PAMS__BADARGLIST`	OpenVMS	Invalid number of arguments supplied.
`PAMS__BADPARAM`	All	Bad argument value.

Table 8-55

Return Code	Platform	Description
`PAMS__INVALIDNUM`	All	Invalid timer number passed to `pams_set_timer`.
`PAMS__INVFORMAT`	All	Invalid timer format specified in the call. Should be P or S.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOTDCL`	All	Process has not been attached to BEA MessageQ.
`PAMS__NOTSUPPORTED`	UNIX Windows NT	The S timer_format is not supported by BEA MessageQ for UNIX and Windows NT systems.
`PAMS__PAMSDOWN`	UNIX Windows NT	The specified BEA MessageQ group is not running.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__RESRCFAIL`	All	Insufficient resources to complete operation.
`PAMS__SUCCESS`	All	Indicates successful completion.

See Also

- `pams_cancel_timer`

Example

Set a Timer

This example shows how to use the BEA MessageQ timer functions by setting a timer to go off every 5 seconds. When the timer expires, it sends messages to itself. While not handling the timer event, it sits and waits for other incoming messages. If it is interrupted, it cancels any outstanding timers. The queue named “`queue_1`” must be defined in your initialization file as a primary queue. The complete code example called `x_timer.c` is contained in the examples directory.

pams_status_text

Receives the severity level and text description of a user-supplied PAMS API return code and moves that information to a user-supplied storage area. If the error code is not known, an error is returned and the call parameters are not filled in.

Syntax

```
int32 pams_status_text ( code, severity, buffer, buflen, retlen)
```

Arguments

Table 8-56

Argument	Data Type	Mechanism	Prototype	Access
`code`	`int32`	reference	`int32 *`	passed
`severity`	`int32`	reference	`int32 *`	returned
`buffer`	`char`	reference	`char *`	returned
`buflen`	`int32`	reference	`int32 *`	passed
`retlen`	`int32`	reference	`int32 *`	returned

Argument Definitions

code

Specifies the return value for which you would like the text description and severity level returned.

severity

Receives a code indicating the severity level of the message. Severity levels apply to both success and error messages. They are designed to provide more information about the message being returned. The valid codes returned to this argument are as follows:

0 = warning
1 = success
2 = error
3 = informational
4 = fatal error

buffer

Receives the text description for the return status supplied.

buflen

Specifies the length of the buffer to store the text description returned. A buffer length of 256 bytes is adequate to store the text description for all return status codes. If the user buffer supplied is large enough, the string is zero terminated. The buffer length must be entered as a positive integer. Supplying a negative integer value to this argument causes the function to return a status of `PAMS__BADPARAM`. If you specify this argument as zero, no text is returned to the buffer and the function returns the status of `PAMS__TRUNCATED`.

retlen

Receives the size of the user-supplied buffer space that was filled by the text description returned.

Description

Application developers use the `pams_status_text` function to obtain a text description and severity level for each API return value. The text description contains both the symbolic name (as it is defined in the include files and described in the documentation) followed by a comma, a space, and then a description of the return value in the following format:

```
PAMS__SUCCESS, normal successful completion
```

In addition to the text description, this function returns a code indicating the severity level for both success and error messages.

For example, `pams_detach_q` has two possible success return codes; `PAMS__SUCCESS` and `PAMS__DETACHED`. The `PAMS__SUCCESS` return code is used to indicate that you successfully detached the specified queue(s). `PAMS__DETACHED` is an informational return code indicating that the call was successful and that you have detached your last queue which effectively detaches your application from the message queuing bus in the same manner as the `pams_exit` function.

Return Values

Table 8-57

Return Code	Platform	Description
PAMS__BADARGLIST	OpenVMS	Invalid number of call parameters specified.
PAMS__BADPARAM	All	Invalid call parameter specified.
PAMS__FAILED	All	There is no translation for the specified return code.

Table 8-57

Return Code	Platform	Description
`PAMS__SUCCESS`	All	Normal successful completion.
`PAMS__TRUNCATED`	All	The description was returned but was truncated.

putil_show_pending

Requests the number of pending messages for a list of selected queues. To use the putil_show_pending function, specify the number of message queues for which you want to obtain a pending message count and the list of queue addresses for which you want to obtain a pending message count. The value returned by this function contains the total number of messages in each memory queue. On OpenVMS systems, this function also returns the number of pending messages in the local recovery journals targeted for delivery to the selected queue.

Syntax

```
int32 putil_show_pending ( count, in_q_list, out_pend_list )
```

Arguments

Table 8-58

Argument	Data Type	Mechanism	Prototype	Access
`count`	`int32`	reference	`int32 *`	passed
`in_q_list`	`short array`	reference	`short array*`	passed
`out_pend_list`	`int32 array`	reference	`int32 array *`	returned

Argument Definitions

count

Supplies the number of queue entries in the **`in_q_list`** argument (the number of indexes in the array). The maximum allowed value is 32,000.

in_q_list

Supplies an array of `int32` values containing the queue numbers for which the pending message count is requested.

out_pend_list

Receives the pending message count for each selected queue.

Return Values

Table 8-59

Return Code	Platform	Description
`PAMS__BADARGLIST`	UNIX Windows NT	Invalid argument supplied to this function.

Table 8-59

Return Code	Platform	Description
`PAMS__BADPARAM`	OpenVMS	Invalid argument supplied to this function.
`PAMS__NETERROR`	Clients	Network error resulted in a communications link abort.
`PAMS__NOTDCL`	All	Process is not attached to BEA MessageQ.
`PAMS__RESRCFAIL`	All	Insufficient resources to complete operation.
`PAMS__PAMSDOWN`	All	The specified BEA MessageQ group is not running.
`PAMS__PREVCALLBUSY`	Clients	Previous call to CLS has not been completed.
`PAMS__SUCCESS`	All	Successful completion.

Example **Display Number of Pending Messages**

This example shows how to use `putil_show_pending` to display the number of pending messages currently in the queue. A queue named “`queue_1`” must be defined during group configuration. The complete code example called `x_shopnd.c` is contained in the examples directory.

CHAPTER

9 Message Reference

This chapter contains detailed descriptions of all BEA MessageQ message-based services alphabetized by message type. Each description lists the message type code name, the name of the BEA MessageQ server performing the service, and a detailed definition of the message area and required arguments to send messages or read response and notification messages using the BEA MessageQ API or scripts. The definition of all BEA MessageQ message-based services messages is now provided in the `p_msg.h` include file.

BEA MessageQ message-based services are sent between a user application program that functions as a requestor and a BEA MessageQ server process that fulfills the request. For messages to be properly understood between systems, message data must be sent and returned in the endian format understood by both the requestor and the server. Most BEA MessageQ message-based services automatically perform this conversion if the endian format of the two systems is different. However, some message-based services do not perform this conversion, therefore, the user application must convert the message to the endian format of the server system to ensure that the message data is correctly interpreted. Each message-based service description notes whether the data structure is RISC aligned and whether the server performs the endian conversion automatically.

AVAIL

Applications can register to receive notification when queues become active or inactive in local and remote groups by sending an AVAIL_REG message to the Avail Server. The AVAIL notification message is sent to registered applications when a queue in the selected group becomes active. See the Obtaining the Status of a Queue topic in the Using Message-Based Services section for an explanation of how to use this message.

Applications must cancel availability notification by sending a message of type AVAIL_DEREG. The application receives a AVAIL_REG_REPLY message indicating the status of the operation. It is important to note that if the distribution queue for an AVAIL registration becomes unavailable, the registration will be automatically deleted by BEA MessageQ. A subsequent attempt to deregister AVAIL services for this distribution queue will result in an error message indicating that the registration does not exist.

Note: The Avail Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _AVAIL {
    q_address target_q;
    } AVAIL;
```

Message Data Fields

Field	Data Type	Script Format	Description
target_q	q_address	DL	Address of queue that is now available.

Arguments

Argument	Data Type	Mechanism	Prototype	Access
Target	Supplied by AVAIL_REG	Supplied by AVAIL_REG	Target	Supplied by AVAIL_REG
Source	AVAIL_SERVER	PAMS_AVAIL_SERVER	Source	AVAIL_SERVER

Argument	Data Type	Mechanism	Prototype	Access
Class	PAMS	MSG_CLAS_ PAMS	Class	PAMS
Type	AVAIL	MSG_TYPE_ AVAIL	Type	AVAIL

See Also

- AVAIL_DEREG
- AVAIL_REG
- AVAIL_REG_REPLY
- UNAVAIL

Example The AVAIL services example illustrates avail services, avail register, avail deregister, and getting avail messages. The complete code example called `x_avail.c` is contained in the examples directory.

AVAIL_DEREG

Applications can register to receive notification when queues become active or inactive in local and remote groups by sending an AVAIL_REG message to the Avail Server. When notification messages are no longer needed, the application sends an AVAIL_DEREG message to the Avail Server to cancel registration. It is important to note that if the distribution queue for an AVAIL registration becomes unavailable, the registration will be automatically deleted by BEA MessageQ. A subsequent attempt to deregister AVAIL services for this distribution queue will result in an error message indicating that the registration does not exist. See the Obtaining the Status of a Queue topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Avail Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _AVAIL_DEREG {
    int16 version;
    int16 filler;
    q_address target_q;
    q_address distribution_q;
    char req_ack;
    } AVAIL_DEREG;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	word	DW	Format version number. Must be 20.
filler	word	DW	Spacing for RISC alignment.
target_q	q_address	DL	Queue being monitored for its availability.
distribution_q	q_address	DL	Queue notified of availability.
req_ack	Boolean	DB	If response required, 1; else 0.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`AVAIL_SERVER`	`PAMS_AVAIL_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`AVAIL_DEREG`	`MSG_TYPE_AVAIL_DEREG`

See Also

- `AVAIL`
- `AVAIL_REG`
- `AVAIL_REG_REPLY`
- `UNAVAIL`

Example

The AVAIL services example illustrates avail services, avail register, avail deregister, and getting avail messages. The complete code example called `x_avail.c` is contained in the examples directory.

AVAIL_REG

Applications can register to receive notification when queues become active or inactive in local and remote groups by sending an AVAIL_REG message to the Avail Server. See the Obtaining the Status of a Queue topic in the Using Message-Based Services section for an explanation of how to use this message. If the application detaches from the distribution queue, the AVAIL registration is automatically deleted. The application must cancel notification, regardless of queue type, by sending a message of type AVAIL_DEREG. The application receives a AVAIL_REG_REPLY message indicating the status of the operation.

Note: The Avail Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _AVAIL_REG {
    int16 version;
    int16 filler;
    q_address target_q;
    q_address distribution_q;
    int32 timeout;
    } AVAIL_REG;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	word	DW	Format version number. Must be 31.
filler	word	DW	Spacing for RISC alignment.
target_q	q_address	DL	Queue to be monitored for availability.
distribution_q	q_address	DL	Queue to receive availability messages.
timeout	int32	DL	Interval (specified in seconds) after which the function should timeout.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`AVAIL_SERVER`	`PAMS_AVAIL_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`AVAIL_REG`	`MSG_TYPE_AVAIL_REG`

See Also

- `AVAIL_REG_REPLY`
- `AVAIL`
- `UNAVAIL`
- `AVAIL_DEREG`

Example

The AVAIL services example illustrates avail services, avail register, avail deregister, and getting avail messages. The complete code example called `x_avail.c` is contained in the examples directory.

AVAIL_REG_REPLY

Applications register to receive notification when queues become active or inactive in local and remote groups by sending an AVAIL_REG message to the Avail Server. The AVAIL_REG_REPLY message indicates whether the application has successfully registered or deregistered from receiving notification messages. See the Obtaining the Status of a Queue topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Avail Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _AVAIL_REG_REPLY {
    int16 status;
    uint16 reg_id;
    int16 number_reg;
    } AVAIL_REG_REPLY;
```

Message Data Fields

Field	**Data Type**	**Script Format**	**Description**
status	word	DW	Status code: 1 = success; 0 = failure.
reg_id	unsigned word	DW	Returned subscription ID.
number_reg	word	DW	Number of registrants left on the Avail list.

Arguments

Argument	**Script Format**	**pams_get_msg Format**
Target	Sender of AVAIL_REG/DEREG	Sender of AVAIL_REG/DEREG
Source	AVAIL_SERVER	PAMS_AVAIL_SERVER
Class	PAMS	MSG_CLAS_PAMS

Argument	Script Format	pams_get_msg Format
Type	`AVAIL_REG_REPLY`	`MSG_TYPE_AVAIL_REG_REPLY`

See Also

- `AVAIL_REG`
- `AVAIL_DEREG`
- `AVAIL`
- `UNAVAIL`

Example

The AVAIL services example illustrates avail services, avail register, avail deregister, and getting avail messages. The complete code example called `x_avail.c` is contained in the examples directory.

DISABLE_NOTIFY

Applications can register to receive notification when cross-group links are established and lost by sending an `ENABLE_NOTIFY` message to the Connect Server. When an application no longer needs to receive notification messages, it deregisters by sending a `DISABLE_NOTIFY` message to the Connect Server. The `DISABLE_NOTIFY` message can stop notification of cross-group link changes. See the Obtain Notification of Cross-Group Links Established and Lost topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Connect Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _ENABLE_NOTIFY {
    char reserved;
    char connection_flag;
    } ENABLE_NOTIFY;
```

Message Data Fields

Field	Data Type	Script Format	Description
`reserved`	unsigned char	DB	Reserved for use by BEA MessageQ.
`connection_flag`	unsigned char	DB	Boolean flag to cancel cross-group connection notification, 1; else 0.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`DISABLE_NOTIFY`	`MSG_TYPE_DISABLE_NOTIFY`

See Also

- `ENABLE_NOTIFY`

- LINK_COMPLETE
- LINK_LOST

DISABLE_Q_NOTIFY_REQ

Applications can register to receive notification when queue states change in local or remote groups by sending an ENABLE_Q_NOTIFY_REQ message. The DISABLE_Q_NOTIFY_REQ is sent to the Queue Server when the application no longer needs to receive notification messages. See the Receiving Attachment Notifications topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Queue Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _Q_NOTIFY_REQ {
    int32 version;
    int32 user_tag;
    } Q_NOTIFY_REQ;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	int32	DL	Version of request.
user_tag	int32	DL	User-specified code to identify this request.

Arguments

Argument	Script Format	pams_get_msg Format
Target	QUEUE_SERVER	PAMS_QUEUE_SERVER
Source	Requesting program	Requesting program
Class	PAMS	MSG_CLAS_PAMS
Type	DISABLE_Q_NOTIFY_REQ	MSG_TYPE_DISABLE_Q_ NOTIFY_REQ

See Also

- DISABLE_Q_NOTIFY_RESP
- ENABLE_Q_NOTIFY_REQ

- `ENABLE_Q_NOTIFY_RESP`
- `Q_UPDATE`

DISABLE_Q_NOTIFY_RESP

Applications can register to receive notification when queue states change in local or remote groups by sending an ENABLE_Q_NOTIFY_REQ message. The DISABLE_Q_NOTIFY_REQ message is sent to the Queue Server when the application no longer needs to receive notification messages. The DISABLE_Q_NOTIFY_RESP message indicates whether the application is successfully deregistered from receiving notification messages. See the Receiving Attachment Notifications topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Queue Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
#define MAX_NUMBER_Q_RECS 50
typedef struct _Q_NOTIFY_RESP {
    int32 version;
    int32 user_tag;
    int32 status_code;
    int32 last_block_flag;
    int32 number_q_recs;
    struct  {
        q_address q_num;
        q_address q_owner;
        int32  q_type;
        int32  q_active_flag;
        int32  q_attached_flag;
        int32  q_owner_pid;
        } q_rec [50];
    } Q_NOTIFY_RESP;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	int32	DL	Version of response.
user_tag	int32	DL	User-specified code from request.
status_code	int32	DL	0=Error 1=Success -2=Refused
last_block_flag	int32	DL	Last block Boolean flag.

Field	Data Type	Script Format	Description
`number_q_recs`	`int32`	DL	Number of records in this message.
`q_num`	`q_address`	DL	Queue number.
`q_owner`	`q_address`	DL	Queue owner (only for secondary queues (SQs)).
`q_type`	`int32`	DL	Queue type (numerically encoded P, S, M).
`q_active_flag`	`int32`	DL	Queue active Boolean flag.
`q_attached_flag`	`int32`	DL	Queue attached Boolean flag.
`q_owner_pid`	`int32`	DL	Queue owner process identification (PID).

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`QUEUE_SERVER`	`PAMS_QUEUE_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`DISABLE_Q_NOTIFY_RESP`	`MSG_TYPE_DISABLE_Q_NOTIFY_RESP`

See Also

- `DISABLE_Q_NOTIFY_REQ`
- `ENABLE_Q_NOTIFY_REQ`
- `ENABLE_Q_NOTIFY_RESP`
- `Q_UPDATE`

ENABLE_NOTIFY

Applications can register to receive `notification` when cross-group links are established and lost by sending an ENABLE_NOTIFY message to the Connect Server. See the Obtain Notification of Cross-Group Links Established and Lost topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Connect Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _ENABLE_NOTIFY {
    char reserved;
    char connection_flag;
    } ENABLE_NOTIFY;
```

Message Data Fields

Field	**Data Type**	**Script Format**	**Description**
`reserved`	unsigned char	DB	Reserved for use by BEA MessageQ.
`connection_flag`	unsigned char	DB	Boolean flag for cross-group connection notification, 1; else 0.

Arguments

Argument	**Script Format**	**pams_get_msg Format**
Target	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`ENABLE_NOTIFY`	`MSG_TYPE_ENABLE_NOTIFY`

See Also

- `DISABLE_NOTIFY`
- `LINK_COMPLETE`
- `LINK_LOST`

ENABLE_Q_NOTIFY_REQ

Applications can register to receive notification when queue states change in local or remote groups by sending an ENABLE_Q_NOTIFY_REQ message. This message requests a list of all active queues and then subsequent notification when queues become attached or detached and active or inactive. See the Receiving Attachment Notifications topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Queue Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _Q_NOTIFY_REQ {
    int32 version;
    int32 user_tag;
    } Q_NOTIFY_REQ;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Version of request.
`user_tag`	`int32`	DL	User-specified code to identify this request.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`QUEUE_SERVER`	`PAMS_QUEUE_SERVER`
Source	Requesting program	Requesting program
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`ENABLE_Q_NOTIFY_REQ`	`MSG_TYPE_ENABLE_Q_NOTIFY_REQ`

See Also

- `DISABLE_Q_NOTIFY_REQ`
- `DISABLE_Q_NOTIFY_RESP`

- `ENABLE_Q_NOTIFY_RESP`
- `Q_UPDATE`

ENABLE_Q_NOTIFY_RESP

Applications can register to receive notification when queue states change in local or remote groups by sending an ENABLE_Q_NOTIFY_REQ message. The ENABLE_Q_NOTIFY_RESP message delivers a list of all active queues and then subsequently notifies the application of attachments, detachments, and changes to active and inactive status using the Q_UPDATE message. See the Receiving Attachment Notifications topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Queue Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
#define MAX_NUMBER_Q_RECS 50
typedef struct _Q_NOTIFY_RESP {
    int32 version;
    int32 user_tag;
    int32 status_code;
    int32 last_block_flag;
    int32 number_q_recs;
    struct  {
        q_address q_num;
        q_address q_owner;
        int32  q_type;
        int32  q_active_flag;
        int32  q_attached_flag;
        int32  q_owner_pid;
        } q_rec [50];
    } Q_NOTIFY_RESP;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	int32	DL	Version of response.
user_tag	int32	DL	User-specified code from request.
status_code	int32	DL	0=Error 1=Success -2=Refused

Field	Data Type	Script Format	Description
`last_block_flag`	`int32`	DL	Last block Boolean flag.
`number_q_recs`	`int32`	DL	Number of records in this message.
`q_num`	`q_address`	DL	Queue number.
`q_owner`	`q_address`	DL	Queue owner (only for secondary queues (SQs)).
`q_type`	`int32`	DL	Queue type (numerically encoded P, S, M).
`q_active_flag`	`int32`	DL	Queue active Boolean flag.
`q_attached_flag`	`int32`	DL	Queue attached Boolean flag.
`q_owner_pid`	`int32`	DL	Queue owner process identification (PID).

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`QUEUE_SERVER`	`PAMS_QUEUE_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`ENABLE_NOTIFY_RESP`	`MSG_TYPE_ENABLE_NOTIFY_RESP`

See Also

- `DISABLE_Q_NOTIFY_REQ`
- `DISABLE_Q_NOTIFY_RESP`
- `ENABLE_Q_NOTIFY_REQ`
- `Q_UPDATE`

LINKMGT_REQ

Applications can use link management messages to explicitly control cross-group connections. Use the LINKMGT_REQ message to request a connection to a remote group, to disconnect from a remote group, or to obtain information about a remote BEA MessageQ group. See the Controlling Cross-Group Links topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Connect Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _TADDRESS {
    int32 len;
    char str [16];
    } TADDRESS;

typedef struct _NODENAME {
     int32 len;
     char str [255];
     } NODENAME;

typedef struct _LINKMGT_REQ {
    int32 version;
    int32 user_tag;
    int32 function_code;
    int32 group_number;
    int32 connect_type;
    int32 reconnect_timer;
    int32 window_size;
    int32 window_delay;
    int32 reserved_space [10];

    TADDRESS transport_addr;
    NODENAME node_name;
    } LINKMGT_REQ;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	int32	DL	Message version.
user_tag	int32	DL	User-specified code to identify this request.

Field	Data Type	Script Format	Description

`function_code`	`int32`	DL	Function of the message using `PSYM_LINKMGT_CMD`:
			`_ENABLE`
			`_DISABLE`
			`_INQUIRY`
			`_CONNECT`
			`_DISCONNECT`

`group_number`	`int32`	DL	Group number to receive action; valid values are between 1 and 32,000; `PSYM_LINKMGT_ALL_GROUPS` indicates all known links.
`connect_type`	`int32`	DL	Type of transport to use, as follows: `PSYM_LINKMGT_TCPIP`

`reconnect_timer`	`int32`	DL	Time it takes for the COM Server to reconnect to a communications link. Enter the number of seconds or the following values:
			`PSYM_LINKMGT_NO_TIMER`
			`PSYM_LINKMGT_USE_PREVIOUS`

`window_size`	`int32`	DL	Size of transmission window (cross-group protocol Version 3.0 and higher). Enter the number of messages or the following value: `PSYM_LINKMGT_USE_PREVIOUS`
`window_delay`	`int32`	DL	Transmission window delay in seconds (cross-group protocol Version 3.0 and higher). Enter the number of seconds or the following value: `PSYM_LINKMGT_USE_PREVIOUS`
`reserved_space`	10-`int32` array	DL(10)	Reserved for BEA MessageQ use.
`transport_addr_len`	`int32`	DL	Length of transport address. Values 0 to 16 bytes; 0 = use previous setting.

Field	Data Type	Script Format	Description
transport_addr	char char str *	A	Transport address string that is 16 bytes in length; the TCP/IP port ID.
node_name_len	int32	DL	Length of node name string; 0 = use previous known value.
node_name	char	A	ASCII text of node name; length determined by node_name_len up to 255 characters.

Arguments

Argument	Script Format	pams_get_msg Format
Target	CONNECT_SERVER	PAMS_CONNECT_SERVER
Source	Requesting program	Requesting program
Class	PAMS	MSG_CLAS_PAMS
Type	LINKMGT_REQ	MSG_TYPE_LINKMGT_REQ

See Also

LINKMGT_RESP

LINKMGT_RESP

Applications can use link management messages to explicitly control cross-group connections. Use the `LINKMGT_REQ` message to request a connection to a remote group, to disconnect from a remote group, or to obtain information about a remote BEA MessageQ group. The `LINKMGT_RESP` message notifies the requesting application if the connection or disconnection request was successful and supplies information about the cross-group connection. See the Controlling Cross-Group Links topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Connect Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _TADDRESS {
    int32 len;
    char str [16];
    } TADDRESS;

typedef struct _NODENAME {
     int32 len;
     char str [255];
     } NODENAME;

typedef struct _LINKMGT_RESP {
    int32 version;
    int32 user_tag;
    int32 status;
    int32 group_number;
    int32 in_link_state;
    int32 out_link_state;
    int32 connect_type;
    int32 platform_id;
    int32 reconnect_timer;
    int32 window_size;
    int32 window_delay;
    int32 reserved_space [10];
    TADDRESS transport_addr;
    NODENAME node_name;
    } LINKMGT_RESP;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	int32	DL	Message version.
user_tag	int32	DL	User-specified code from request.
status	int32	DL	Completion status
group_number	int32	DL	Group number to receive action. Valid values are between 1 and 32,000; PSYM_LINKMGT_ALL_GROUPS indicates all known links.
in_link_state	int32	DL	State of inbound link at time of request. Values are: PSYM_LINKMGT_UNKNOWN PSYM_LINKMGT_NOCNT PSYM_LINKMGT_CONNECTED PSYM_LINKMGT_DISABLED
out_link_state	int32	DL	State of outbound link at time of request; same values as in_link_state.
connect_type	int32	DL	Type of transport to use as follows: PSYM_LINKMGT_TCPIP

Field	Data Type	Script Format	Description

`platform_id`	`int32`	DL	Platform type preceded by the prefix `PSYM_PLATFORM`. Valid values are:
			`VAX_VMS`
			`VAX_ULTRIX`
			`RISC_ULTRIX`
			`HP9000_HPUX`
			`MOTOROLA_VR32`
			`SPARC_SUNOS`
			`IBM_RS6000_AIX`
			`OS2`
			`MSDOS`
			`PDP11_RSX`
			`VAXELN`
			`MACINTOSH`
			`SCO_UNIX`
			`M68K`
			`VMS_AXP`
			`UNIX`
			`WINDOWSNT`
			`OSF1_AXP`
			`DYNIX_X86`
			`UNKNOWN`

`reconnect_timer`	`int32`	DL	Time it takes for the COM Server to reconnect to a communications link. Enter the number of seconds or the following values:
			`PSYM_LINKMGT_NO_TIMER`
			`PSYM_LINKMGT_USE_PREVIOUS`

`window_size`	`int32`	DL	Size of transmission window (cross-group protocol Version 3.0 and higher).
`window_delay`	`int32`	DL	Transmission window delay in seconds (cross-group protocol Version 3.0 and higher).
`reserved_space`	`10-int32 array`	DL(10)	Reserved for BEA MessageQ use.
`transport_addr_len`	`int32`	DL	Length of transport address. Values 0 to 16 bytes; 0 = use previous setting.

Field	Data Type	Script Format	Description
`transport_addr`	`char`	A	Transport address string 16 bytes in length, the TCP/IP port ID.
`node_name_len`	`int32`	DL	Length of node name string. 0 = use previous known value.
`node_name`	`char`	A	ASCII text of node name; length determined by `node_name_len` up to 255 characters.

Status Codes

Status Code	Description
`PSYM_LINKMGT_ALREADYUP`	Link already connected.
`PSYM_LINKMGT_MSGCONTENT`	Message incomplete or content inconsistent with dialog.
`PSYM_LINKMGT_MSGFMT`	Format error in dialog.
`PSYM_LINKMGT_NOGROUP`	Group is unknown.
`PSYM_LINKMGT_NOPRIV`	No privilege for attempted operation.
`PSYM_LINKMGT_NOTRANSPORT`	Requested transport is not available.
`PSYM_LINKMGT_NOTSUPPORTED`	Feature not supported.
`PSYM_LINKMGT_OPERATIONFAIL`	Requested operation failed.
`PSYM_LINKMGT_SUCCESS`	Normal successful return.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`

Argument	Script Format	pams_get_msg Format
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LINKMGT_RESP`	`MSG_TYPE_LINKMGT_RESP`

See Also

- `LINKMGT_REQ`

LINK_COMPLETE

Applications can register to receive notification when cross-group links are established and lost by sending an `ENABLE_NOTIFY` message to the Connect Server. Registered applications receive a `LINK_COMPLETE` message each time a cross-group connection occurs. See the Obtain Notification of Cross-Group Links Established and Lost topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Connect Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
typedef struct _LINK_NOTIFICATION {
    int16 group_number;
    int16 filler1;
    char  os_type;
    char  filler2;
    } LINK_NOTIFICATION;
```

Message Data Fields

Field	Data Type	Script Format	Description
`group_number`	`word`	DW	Group address associated with link.
`filler1`	`word`	DW	Reserved for BEA MessageQ.
`os_type`	`byte`	A(1)	Code indicating operating system of remote node.
`filler2`	`byte`	XB	Reserved for BEA MessageQ.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`

Argument	Script Format	pams_get_msg Format
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LINK_COMPLETE`	`MSG_TYPE_LINK_COMPLETE`

See Also

- `DISABLE_NOTIFY`
- `ENABLE_NOTIFY`
- `LINK_LOST`

LINK_LOST

Applications can register to receive notification when cross-group links are established and lost by sending an ENABLE_NOTIFY message to the Connect Server. Registered applications receive a LINK_LOST message each time a cross-group connection is lost. See the Obtain Notification of Cross-Group Links Established and Lost topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Connect Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
typedef struct _LINK_NOTIFICATION {
    int16 group_number;
    int16 filler1;
    char  os_type;
    char  filler2;
    } LINK_NOTIFICATION;
```

Message Data Fields

Field	Data Type	Script Format	Description
group_number	word	DW	Group address associated with link.
filler1	word	DW	Reserved for BEA MessageQ.
os_type	byte	A(1)	Code indicating operating system of remote node.
filler2	byte	XB	Reserved for BEA MessageQ.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	CONNECT_SERVER	PAMS_CONNECT_SERVER

Argument	Script Format	pams_get_msg Format
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LINK_LOST`	`MSG_TYPE_LINK_LOST`

See Also

- `DISABLE_NOTIFY`
- `ENABLE_NOTIFY`
- `LINK_COMPLETE`

LIST_ALL_CONNECTIONS (Request)

An application can request a listing of all active and configured cross-group connections by sending a `LIST_ALL_CONNECTIONS` message to the Connect Server. The reply to this request is a variable-length message of the same type and class containing the cross-group connection information. See the Listing Cross-Group Connections, Entries, and Groups topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: This message is RISC aligned.

C Message Structure

None.

Message Data Fields

None.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`
Source	Requesting program	Requesting program
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LIST_ALL_CONNECTIONS`	`MSG_TYPE_LIST_ALL_CONNECTIONS`

See Also

- `LIST_ALL_CONNECTIONS` response message
- `LIST_ALL_ENTRIES` (Request)
- `LIST_ALL_ENTRIES` (Response)
- `LIST_ALL_GROUPS` (Request)
- `LIST_ALL_GROUPS` (Response)

LIST_ALL_CONNECTIONS (Response)

An application can request a listing of all active and configured cross-group connections by sending a LIST_ALL_CONNECTIONS message to the Connect Server. The reply to this request is a variable length-message of the same type and class containing the cross-group connection information. To read the information returned, the application must total the number of bytes in the reply and divide by the cross-group entry length, which is 20 bytes, to determine the number of records returned. See the Listing Cross-Group Connections, Entries, and Groups topic in the Using Message-Based Services section for an explanation of how to use this message.

This message does not return any information on groups with no link connection. The state field for LIST_ALL_CONNECTIONS should always be 3 (linked).

Note: The Connect Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
typedef struct _GROUP_RECORD {
    int16 group_number;
    char group_name[4];
    char uic[3];
    char os_type;
    char node[6];
    char state;
    char reserved[3];
    } GROUP_RECORD;
```

Message Data Fields

Field	Data Type	Script Format	Description
group_number	word	DW	Group address number.
group_name	4-char array	A(4)	Name truncated to 4 characters.
uic	3-char array	A(3)	Octal group user identification code (UIC).
os_type	char	A(1)	Operating system type of group.

Field	Data Type	Script Format	Description
`node`	6-`char` array	A(6)	Network node name.
`state`	`char`	A(1)	1=No link 2=Pending 3=Linked
`reserved`	3-`char`	ZB 3	Reserved for BEA MessageQ.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Source	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LIST_ALL_CONNECTIONS`	`MSG_TYPE_LIST_ALL_ CONNECTIONS`

See Also

- `LIST_ALL_CONNECTIONS` request message
- `LIST_ALL_ENTRIES` (Request)
- `LIST_ALL_ENTRIES` (Response)
- `LIST_ALL_GROUPS` (Request)
- `LIST_ALL_GROUPS` (Response)

LIST_ALL_ENTRIES (Request)

An application can request a listing of all attached and configured queues in a group by sending a `LIST_ALL_ENTRIES` message to the Connect Server. The reply to this request is a variable-length message of the same type and class containing the queue information. See the Listing Cross-Group Connections, Entries, and Groups topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: This message is RISC aligned.

C Message Structure

None.

Message Data Fields

None.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LIST_ALL_ENTRIES`	`MSG_TYPE_LIST_ALL_ ENTRIES`

See Also

- `LIST_ALL_ENTRIES` response message
- `LIST_ALL_CONNECTIONS` (Request)
- `LIST_ALL_CONNECTIONS` (Response)
- `LIST_ALL_GROUPS` (Request)
- `LIST_ALL_GROUPS` (Response)

LIST_ALL_ENTRIES (Response)

An application can request a listing of all attached and configured queues in a group by sending a `LIST_ALL_ENTRIES` message to the Connect Server. The reply to this request is a variable length message of the same type and class containing the queue information. To read the information returned, the application must total the number of bytes in the reply and divide by the queue entry length, which is 24 bytes, to determine the number of records returned. See the Listing Cross-Group Connections, Entries, and Groups topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Connect Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
typedef struct _QLIST_RECORD {
    char q_name [20];
    int16 q_number;
    char attach_flag;
    char reserved;
    } QLIST_RECORD;
```

Message Data Fields

Field	**Data Type**	**Script Format**	**Description**
`q_name`	20-`char` array	A(20)	Queue name, truncated to fit.
`q_number`	`word`	DW	Local queue address number.
`attach_flag`	Boolean	DB	1=Attached 0=Unattached
`reserved`	`byte`	ZB	Reserved for BEA MessageQ.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LIST_ALL_ENTRIES`	`MSG_TYPE_LIST_ALL_ENTRIES`

See Also

- `LIST_ALL_ENTRIES` request message
- `LIST_ALL_GROUPS` (Request)
- `LIST_ALL_GROUPS` (Response)
- `LIST_ALL_CONNECTIONS` (Request)
- `LIST_ALL_CONNECTIONS` (Response)

LIST_ALL_GROUPS (Request)

An application can request a listing of all groups on a message queuing bus by sending a LIST_ALL_GROUPS message to the Connect Server. The reply to this request is a variable-length message of the same type and class containing the group information. See the Listing Cross-Group Connections, Entries, and Groups topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: This message is RISC aligned.

C Message Structure

None.

Message Data Fields

None.

Arguments

Argument	Script Format	pams_get_msg Format
Target	CONNECT_SERVER	PAMS_CONNECT_SERVER
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	PAMS	MSG_CLAS_PAMS
Type	LIST_ALL_GROUPS	MSG_TYPE_LIST_ALL_GROUPS

See Also

- LIST_ALL_GROUPS response message
- LIST_ALL_CONNECTIONS (Request)
- LIST_ALL_CONNECTIONS (Response)
- LIST_ALL_ENTRIES (Request)
- LIST_ALL_ENTRIES (Response)

LIST_ALL_GROUPS (Response)

An application can request a listing of all groups, connected and unconnected, on a message queuing bus by sending a `LIST_ALL_GROUPS` message to the Connect Server. The reply to this request is a variable-length message of the same type and class containing the group information. To read the information returned, the application must total the number of bytes in the reply and divide by the group entry length, which is 18 bytes, to determine the number of records returned. See the Listing Cross-Group Connections, Entries, and Groups topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Connect Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
typedef struct _LIST_ALL_RESP {
    int16 group_number;
    char  group_name [4];
    char  uic_number [3];
    char  operating_system;
    char  decnet_node [6];
    char  connection_state;
    char reserved[3];
    } LIST_ALL_RESP;
```

Message Data Fields

Field	Data Type	Script Format	Description
`group_number`	`word`	DW	Group address number.
`group_name`	`4-char` array	A(4)	Name truncated to 4 characters.
`uic_number`	`3-char` array	A(3)	Octal group user identification code (UIC).
`operating_system`	`char`	A(1)	Operating system type of group.

Field	Data Type	Script Format	Description
`decnet_node`	6-`char` array	A(6)	Current DECnet node name. This can also be the TCP/IP node name. TCP/IP node names longer than 6 characters are truncated.
`connection_state`	`char`	A(1)	1=No link 2=Pending 3=Linked
`reserved`	3-`char` (VMS) 1-`char` (UNIX)	ZB	Reserved for BEA MessageQ.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`CONNECT_SERVER`	`PAMS_CONNECT_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LIST_ALL_GROUPS`	`MSG_TYPE_LIST_ALL_GROUPS`

See Also

- `LIST_ALL_GROUPS` request message
- `LIST_ALL_CONNECTIONS` (Request)
- `LIST_ALL_CONNECTIONS` (Response)
- `LIST_ALL_ENTRIES` (Request)
- `LIST_ALL_ENTRIES` (Response)

LIST_ALL_Q_REQ

The `LIST_ALL_Q_REQ` message is sent to the Queue Server to request a list of all attached permanent and temporary queues for a local or remote group. See the Listing Attached Queues in a Group topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Queue Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _Q_NOTIFY_REQ {
    int32 version;
    int32 user_tag;
    } Q_NOTIFY_REQ;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Version of request.
`user_tag`	`int32`	DL	User-specified code to identify this request.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`QUEUE_SERVER`	`PAMS_QUEUE_SERVER`
Source	Requesting program	Requesting program
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LIST_ALL_Q_REQ`	`MSG_TYPE_LIST_ALL_Q_REQ`

See Also

- `LIST_ALL_Q_RESP`

LIST_ALL_Q_RESP

The `LIST_ALL_Q_RESP` message provides a list of all permanent queues and all attached temporary queues for a local or remote group. This information is requested by sending a `LIST_ALL_Q_REQ` message to the Queue Server. Because the response message may contain a long list of queue names, the application must allocate a sufficient buffer size to store the information returned. See Listing Attached Queues in a Group in Chapter 5, "Using Message-Based Services" for an explanation of how to use this message.

Note: The Queue Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
#define MAX_NUMBER_Q_RECS 50
typedef struct _Q_NOTIFY_RESP {
    int32 version;
    int32 user_tag;
    int32 status_code;
    int32 last_block_flag;
    int32 number_q_recs;
    struct  {
        q_address q_num;
        q_address q_owner;
        int32  q_type;
        int32  q_active_flag;
        int32  q_attached_flag;
        int32  q_owner_pid;
        } q_rec [50];
    } Q_NOTIFY_RESP;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Version of response.
`user_tag`	`int32`	DL	User-specified code from request.
`status_code`	`int32`	DL	0=Error 1=Success -2=Refused

Field	**Data Type**	**Script Format**	**Description**
`last_block_flag`	`int32`	DL	Last block Boolean flag.
`number_q_recs`	`int32`	DL	Number of records in this message.
`q_num`	`q_address`	DL	Queue number.
`q_owner`	`q_address`	DL	Queue owner (only for secondary queues (SQs)).
`q_type`	`int32`	DL	Queue type (numerically encoded `P`, `S`, `M`).
`q_active_flag`	`int32`	DL	Queue active Boolean flag.
`q_attached_flag`	`int32`	DL	Queue attached Boolean flag.
`q_owner_pid`	`int32`	DL	Queue owner process identification (PID). On Windows NT systems, thread identifier is returned.

Arguments

Argument	**Script Format**	**pams_get_msg Format**
Target	Requesting program	Requesting program
Source	`QUEUE_SERVER`	`PAMS_QUEUE_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LIST_ALL_Q_RESP`	`MSG_TYPE_LIST_ALL_Q_RESP`

See Also

- `LIST_ALL_Q_REQ`

LOCATE_Q_REP

The pams_locate_q function requests the queue address for a queue name. When this function is performed asynchronously, the results are returned in the `LOCATE_Q_REP` message. This message provides the location in the search list where the name is found, the status of the operation, a tag that can be set by the user, and the queue address associated with the name.

Note: This message is RISC aligned.

C Message Structure

```
typedef struct _LOCATE_Q_REP {
    int32 version;
    int32 search_loc;
    q_address object_handle;
    int32 status;
    int32 trans_id;
    char q_name [256];
    } LOCATE_Q_REP;
```

Message Data Fields

Field	**Data Type**	**Script Format**	**Description**
`version`	`int32`	DL	Format version number.
`search_loc`	`int32`	DL	Location in which name is found.
`object_handle`	`q_address`	DL	Queue address associated with name.
`status`	`int32`	DL	Return code from `pams_locate_q`.
`trans_id`	`int32`	DL	User-specified tag.
`q_name`	256-character array	A(256)	Name to locate.

Arguments

Argument	**Script Format**	**pams_get_msg Format**
Target	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ

Argument	Script Format	pams_get_msg Format
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`LOCATE_Q_REP`	`MSG_TYPE_LOCATE_Q_REP`

MRS_ACK

The MRS_ACK message acknowledges the delivery of a recoverable message at the delivery interest point when a nonblocking request is issued. It responds to a pams_put_msg request when delivery modes of `PDEL_MODE_AK_DQF`, `PDEL_MODE_AK_SAF`, or `PDEL_MODE_AK_CONF` are specified. Status codes for the send operation are extracted from the PAMS Status Block (PSB), an argument value which is returned to the `pams_get_msg`, `pams_get_msga`, and `pams_get_msgw` function when the recoverable message is read. The status codes for the **`psb`** and **`uma`** arguments are listed in the Status Codes section of this description.

Note: This message is RISC aligned.

C Message Structure

None.

Message Data Fields

None.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Sender program	Sender program
Source	`MRS_SERVER`	`PAMS_MRS_SERVER`
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_ACK`	`MSG_TYPE_MRS_ACK`

Status Code

Message	PSB Status
`PAMS__DQF_DEVICE_FAIL`	Message is not recoverable; destination queue file (DQF) I/O failed.
`PAMS__ENQUEUED`	Message is recoverable.
`PAMS__MRS_RES_EXH`	Message is not recoverable; MRS resource exhaustion.

Message	PSB Status
PAMS__NO_DQF	Message is not recoverable; no DQF for target queue.
PAMS__NO_SAF	Message is not recoverable; no SAF file for target queue.
PAMS__SAF_DEVICE_FAIL	Message is not recoverable; SAF I/O failed.
PAMS__SAF_FORCED	Message is written to SAF file to maintain first-in/first-out (FIFO) order.
PAMS__SENDER_TMOEXPIRED	Send timeout expired prior to completion of MRS actions.
PAMS__STORED	Message is recoverable in store and forward (SAF) file. (Delivery mode was PDEL_MODE_AK_SAF.)

UMA Status

Message	UMA Status
PAMS__DISC_SUCCESS	Message is not recoverable in DQF; UMA was PDEL_UMA_DISC; message discarded.
PAMS__DISC_FAILED	Message is not recoverable in DQF; UMA was PDEL_UMA_DISC; message could not be discarded.
PAMS__DISCL_SUCCESS	Message is not recoverable in DQF; UMA was PDEL_UMA_DISC; message discarded after logging recoverability failure.
PAMS__DISCL_FAILED	Message is not recoverable in DQF; UMA was PDEL_UMA_DISC; recoverability failure could not be logged or message could not be discarded.
PAMS__DLJ_SUCCESS	Message is not recoverable in DQF; UMA was PDEL_UMA_DLJ; message written to dead letter journal (DLJ).
PAMS__DLJ_FAILED	Message is not recoverable in DQF; UMA was PDEL_UMA_DLJ; dead letter journal write failed.

Message	UMA Status
`PAMS__DLQ_SUCCESS`	Message is not recoverable in DQF; UMA was `PDEL_UMA_DLQ`; message queued to dead letter queue.
`PAMS__DLQ_FAILED`	Message is not recoverable in DQF; UMA was `PDEL_UMA_DLQ`; message could not be queued to dead letter queue.
`PAMS__NO_UMA`	Message is recoverable; undeliverable message action (UMA) not executed.
`PAMS__RTS_SUCCESS`	Message is not recoverable in DQF; UMA was `PDEL_UMA_RTS`; message returned to sender.
`PAMS__RTS_FAILED`	Message is not recoverable in DQF; UMA was `PDEL_UMA_RTS`; message could not be returned to sender.
`PAMS__SAF_SUCCESS`	Message is not recoverable in DQF; UMA was `PDEL_UMA_SAF`; message recoverable from SAF file.
`PAMS__SAF_FAILED`	Message is not recoverable in DQF; UMA was `PDEL_UMA_SAF`; SAF write failed.

MRS_DQF_SET

Applications can request to open, close, or fail over a destination queue file (DQF) by sending an MRS_DQF_SET message to the MRS Server. The failover function renames a DQF file, associating it with another target queue that does not currently have a DQF associated with it. See the Opening, Closing, and Failing Over SAF and DQF Files topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is sent between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
/*****************************************/
/* ACTION VALUES FOR MRS_DQF_SET message */
/*****************************************/
#define DQF_SET_OPEN 1
#define DQF_SET_CLOSE 2
#define DQF_SET_FAILOVER 3

/*****************************************/
/* STATUS VALUES FOR MRS_DQF_SET message */
/*****************************************/
#define DQF_SET_ERROR 0
#define DQF_SET_SUCCESS 1
#define DQF_SET_REFUSED 2

typedef struct _MRS_DQF_SET {
    int16 version;
    int16 action;
    int32 status;
    q_address original_target;
    q_address new_target;
    int32 original_mrs_area_len;
    char original_mrs_area [256];
    } MRS_DQF_SET;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number. Must be 0.
`action`	`word`	DW	1 = Open 2 = Close 3 = Fail over
`status`	`int32`	DL	0 = Error 1 = Success 2 = Refused
`original_target`	`q_address`	DL	Queue address of DQF.
`new target`	`q_address`	DL	Queue address of new DQF for failover.
`original_mrs_area_len`	`int32`	DL	Number of bytes in original MRS area specification for failover.
`original_mrs_area`	256-`byte` array	A(256)	MRS original area specification for failover.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`MRS_SERVER`	`PAMS_MRS_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_DQF_SET`	`MSG_TYPE_MRS_DQF_SET`

See Also

- `MRS_DQF_SET_REP`
- `MRS_SAF_SET`
- `MRS_SAF_SET_REP`

MRS_DQF_SET_REP

Applications can request to open, close, or fail over a destination queue file (DQF) by sending an MRS_DQF_SET message to the MRS Server. The failover function renames a DQF file, associating it with another target queue that does not currently have a DQF associated with it. The MRS_DQF_SET_REP message returns the status of the request. See the Opening, Closing, and Failing Over SAF and DQF Files topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
/*****************************************/
/* ACTION VALUES FOR MRS_DQF_SET message */
/*****************************************/
#define DQF_SET_OPEN 1
#define DQF_SET_CLOSE 2
#define DQF_SET_FAILOVER 3

/*****************************************/
/* STATUS VALUES FOR MRS_DQF_SET message */
/*****************************************/
#define DQF_SET_ERROR 0
#define DQF_SET_SUCCESS 1
#define DQF_SET_REFUSED 2

typedef struct _MRS_DQF_SET {
    int16 version;
    int16 action;
    int32 status;
    q_address original_target;
    q_address new_target;
    int32 original_mrs_area_len;
    char original_mrs_area [256];
    } MRS_DQF_SET;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number. Must be 0.
`action`	`word`	DW	1 = Open 2 = Close 3 = Fail over
`status`	`int32`	DL	0 = Error 1 = Success 2 = Refused
`original_ target`	`q_address`	DL	Queue address of DQF.
`new_target`	`q_address`	DL	Queue address of new DQF for failover.
`original_mrs_area_len`	`int32`	DL	Number of bytes in original MRS area specification for failover.
`original_mrs_area`	256-`byte` array	A(256)	MRS original area specification for failover.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`MRS_SERVER`	`PAMS_MRS_SERVER`
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_DQF_SET_REP`	`MSG_TYPE_MRS_DQF_SET_REP`

See Also

- `MRS_DQF_SET`
- `MRS_SAF_SET`
- `MRS_SAF_SET_REP`

MRS_DQF_TRANSFER

Applications can request the transfer of the contents of one DQF to another by sending a `MRS_DQF_TRANSFER` message to the Qtransfer Server. Using this failover method, when a node fails, the Qtransfer Server can transfer messages from a recoverable queue on a node that has failed to a recoverable queue on a node that is currently processing messages. See the Transferring the Contents of a Destination Queue File topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The Qtransfer Server **does not** perform endian conversion when this message is sent between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted.

C Message Structure

```
typedef struct _MRS_DQF_TRANSFER {
    int16 version;
    int32 user_tag;
    int16 status;
    int32 send_count;
    int16 from_dqf_len;
    char from_dqf_file [256];
    int16 to_q;
    } MRS_DQF_TRANSFER;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number.
`user_tag`	`int32`	XL	User-defined tag.
`status`	`word`	DW	Not used.
`send_count`	`int32`	DL	Count of successful transfers.
`from_dqf_len`	`word`	DW	Number of bytes in DQF file specification.
`from_dqf_file`	256-byte array	A(256)	File specification of DQF.
`to_q`	`word`	DW	Local address of queue to receive transfer.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`QTRANSFER`	`PAMS_QTRANSFER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_DQF_TRANSFER`	`MSG_TYPE_MRS_DQF_TRANSFER`

See Also

- `MRS_DQF_TRANSFER_REP`
- `MRS_DQF_TRANSFER_ACK`

MRS_DQF_TRANSFER_ACK

Applications can request the transfer of the contents of one DQF file to another by sending an `MRS_DQF_TRANSFER` message to the Qtransfer Server. Using this failover method, when a node fails, the Qtransfer Server can transfer messages from a recoverable queue on a node that has failed to a recoverable queue on a node that is currently processing messages. The `MRS_DQF_TRANSFER_ACK` message is returned to the sender to acknowledge the receipt of the request. See the Transferring the Contents of a Destination Queue File topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The Qtransfer Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted.

C Message Structure

```
typedef struct _MRS_DQF_TRANSFER {
    int16 version;
    int32 user_tag;
    int16 status;
    int32 send_count;
    int16 from_dqf_len;
    char from_dqf_file [256];
    int16 to_q;
    } MRS_DQF_TRANSFER;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number.
`user_tag`	`int32`	XL	User-defined tag.
`status`	`word`	DW	0=Error 1=Success 2=Refused
`send_count`	`int32`	DW	Count of successful transfers.

Field	Data Type	Script Format	Description
`from_dqf_len`	`word`	DW	Number of bytes in DQF file specification.
`from_dqf_file`	256-byte array	A(256)	File specification of DQF file to read.
`to_q`	`word`	DW	Local address of queue to receive transfer.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`QTRANSFER`	`PAMS_QTRANSFER`
Source	Supplied by BEA MessageQ	Supplied by MessaqeQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_DQF_TRANSFER_ACK`	`MSG_TYPE_DQF_TRANSFER_ ACK`

See Also

- `MRS_DQF_TRANSFER`
- `MRS_DQF_TRANSFER_REP`

MRS_DQF_TRANSFER_REP

Applications can request the transfer of the contents of one destination queue file to another by sending an MRS_DQF_TRANSFER message to the Qtransfer Server. Using this failover method, when a node fails, the Qtransfer Server can transfer messages from a recoverable queue on a node that has failed to a recoverable queue on a node that is currently processing messages. The MRS_DQF_TRANSFER_REP message is returned to the sender to indicate the completion status of the request. See the Transferring the Contents of a Destination Queue File topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The Qtransfer Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted.

C Message Structure

```
typedef struct _MRS_DQF_TRANSFER {
    int16 version;
    int32 user_tag;
    int16 status;
    int32 send_count;
    int16 from_dqf_len;
    char from_dqf_file [256];
    int16 to_q;
    } MRS_DQF_TRANSFER;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number.
`user_tag`	`int32`	XL	User-defined tag.
`status`	`word`	DW	0=Error 1=Success 2=Refused
`send_count`	`int32`	DL	Count of successful transfers.
`from_dqf_len`	`word`	DW	Number of bytes DQF file specification.

Field	Data Type	Script Format	Description
`from_dqf_file`	256-byte array	A(256)	File specification of DQF file to read.
`to_q`	`word`	DW	Local address of queue to receive transfer.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`QTRANSFER`	`PAMS_QTRANSFER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_DQF_TRANSFER_REP`	`MSG_TYPE_MRS_DFQ_ TRANSFER_REP`

See Also

- `MRS_DQF_TRANSFER`
- `MRS_DQF_TRANSFER_ACK`

MRS_JRN_DISABLE

Disables journaling for a running message queuing group. This service is used to disable journaling before failing over auxiliary journals. See the Controlling Journaling to the PCJ File topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is sent between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
/*****************************************/
/* STATUS VALUES FOR JRN_ENABLE message   */
/*****************************************/
#define JRN_SET_ERROR 0
#define JRN_SET_SUCCESS 1
#define JRN_SET_REFUSED 2
#define JRN_SET_ALREADY_DISABLED 3
#define JRN_SET_ALREADY_ENABLED 4
#define JRN_SET_SERVER_NOTUP 5


typedef struct _MRS_JRN_SET_ALL {
    int32 version;
    int32 dqf_status;
    int32 saf_status;
    int32 pcj_status;
    int32 dlj_status;
    } MRS_JRN_SET_ALL;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Format version number. Must be 0.
`dqf_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 3 = Already Disabled

Field	Data Type	Script Format	Description
`saf_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 3 = Already Disabled
`pcj_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 3 = Already Disabled 5 = Server Not Available
`dlj_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 3 = Already Disabled 5 = Server Not Available

Arguments

Argument	Script Format	pams_get_msg Format
Target	`MRS_SERVER`	`PAMS_MRS_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_JRN_DISABLE`	`MSG_TYPE_MRS_JRN_DISABLE`

See Also

- `MRS_JRN_DISABLE_REP`
- `MRS_JRN_ENABLE`
- `MRS_JRN_ENABLE_REP`

MRS_JRN_DISABLE_REP

Applications can request to disable journaling for a running message queuing group by sending an `MRS_JRN_DISABLE` message to the MRS Server. The `MRS_JRN_DISABLE_REP` message returns the status of the request. This service is used before failing over auxiliary journals. See the Controlling Journaling to the PCJ File topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
/*****************************************/
/* STATUS VALUES FOR JRN_ENABLE message   */
/*****************************************/
#define JRN_SET_ERROR 0
#define JRN_SET_SUCCESS 1
#define JRN_SET_REFUSED 2
#define JRN_SET_ALREADY_DISABLED 3
#define JRN_SET_ALREADY_ENABLED 4
#define JRN_SET_SERVER_NOTUP 5


typedef struct _MRS_JRN_SET_ALL {
    int32 version;
    int32 dqf_status;
    int32 saf_status;
    int32 pcj_status;
    int32 dlj_status;
    } MRS_JRN_SET_ALL;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Format version number. Must be 0.
`dqf_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 3 = Already Disabled

Field	Data Type	Script Format	Description
`saf_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 3 = Already Disabled
`pcj_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 3 = Already Disabled 5 = Server Not Available
`dlj_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 3 = Already Disabled 5 = Server Not Available

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`MRS_SERVER`	`PAMS_MRS_SERVER`
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_JRN_DISABLE_REP`	`MSG_TYPE_MRS_JRN_` `DISABLE_REP`

See Also

- `MRS_JRN_DISABLE`
- `MRS_JRN_ENABLE`
- `MRS_JRN_ENABLE_REP`

MRS_JRN_ENABLE

Enables journaling for a running message queuing group after it has been disabled using the `MRS_JRN_DISABLE` message. This service is used before failing over auxiliary journals. See the Controlling Journaling to the PCJ File topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is sent between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
/*****************************************/
/* STATUS VALUES FOR JRN_ENABLE message   */
/*****************************************/
#define JRN_SET_ERROR 0
#define JRN_SET_SUCCESS 1
#define JRN_SET_REFUSED 2
#define JRN_SET_ALREADY_DISABLED 3
#define JRN_SET_ALREADY_ENABLED 4
#define JRN_SET_SERVER_NOTUP 5

typedef struct _MRS_JRN_SET_ALL {
    int32 version;
    int32 dqf_status;
    int32 saf_status;
    int32 pcj_status;
    int32 dlj_status;
    } MRS_JRN_SET_ALL;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Format version number. Must be 0.
`dqf_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 4 = Already Enabled

Field	Data Type	Script Format	Description
`saf_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 4 = Already Enabled
`pcj_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 4 = Already Enabled 5 = Server Not Available
`dlj_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 4 = Already Enabled 5 = Server Not Available

Arguments

Argument	Script Format	pams_get_msg Format
Target	`MRS_SERVER`	`PAMS_MRS_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_JRN_ENABLE`	`MSG_TYPE_MRS_JRN_ENABLE`

See Also

`MRS_JRN_DISABLE`

`MRS_JRN_DISABLE_REP`

`MRS_JRN_ENABLE_REP`

MRS_JRN_ENABLE_REP

Applications can request to reenable journaling for a running message queuing group after it has been disabled by sending an `MRS_JRN_ENABLE` message to the MRS Server. The `MRS_JRN_ENABLE_REP` message returns the status of the request. This service is used with MRS before failing over auxiliary journals. See the Controlling Journaling to the PCJ File topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
/*******************************************/
/* STATUS VALUES FOR JRN_ENABLE message   */
/*******************************************/
#define JRN_SET_ERROR 0
#define JRN_SET_SUCCESS 1
#define JRN_SET_REFUSED 2
#define JRN_SET_ALREADY_DISABLED 3
#define JRN_SET_ALREADY_ENABLED 4
#define JRN_SET_SERVER_NOTUP 5

typedef struct _MRS_JRN_SET_ALL {
    int32 version;
    int32 dqf_status;
    int32 saf_status;
    int32 pcj_status;
    int32 dlj_status;
    };
typedef struct MRS_JRN_SET_ALL;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Format version number. Must be 0.
`dqf_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 4 = Already Enabled

Field	Data Type	Script Format	Description
`saf_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 4 = Already Enabled
`pcj_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 4 = Already Enabled 5 = Server Not Available
`dlj_status`	`int32`	DL	0 = Error 1 = Success 2 = Refused 4 = Already Enabled 5 = Server Not Available

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`MRS_SERVER`	`PAMS_MRS_SERVER`
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_JRN_ENABLE_REP`	`MSG_TYPE_MRS_JRN_ENABLE_REP`

See Also

`MRS_JRN_DISABLE`

`MRS_JRN_DISABLE_REP`

`MRS_JRN_ENABLE`

MRS_SAF_SET

Applications can request to open, close, or failover (redirect) a store-and-forward file (SAF) by sending an `MRS_SAF_SET` message to the MRS Server. The failover function renames a SAF file, associating it with another target queue that does not currently have a SAF associated with it. See the Opening, Closing, and Failing Over SAF and DQF Files topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is sent between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
/****************************************/
/* ACTION VALUES FOR MRS_SAF_SET message */
/****************************************/
#define SAF_SET_OPEN 4
#define SAF_SET_CLOSE 5
#define SAF_SET_FAILOVER 6


/****************************************/
/* STATUS VALUES FOR MRS_SAF_SET message */
/****************************************/
#define JRN_SET_ERROR 0
#define JRN_SET_SUCCESS 1
#define JRN_SET_REFUSED 2


typedef struct _MRS_SAF_SET {
    int16 version;
    int16 action;
    int32 status;
    q_address original_target;
    q_address new_target;
    int32 original_mrs_area_len;
    char original_mrs_area [256];
    int16 original_owner_group;
    int16 new_owner_group;
    } MRS_SAF_SET;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number. Must be 0.
`action`	`word`	DW	4 = Open 5 = Close 6 = Failover
`status`	`int32`	DL	0 = Error 1 = Success 2 = Refused
`original_target`	`q_address`	DL	Queue address of SAF.
`new_target`	`q_address`	DL	Queue address of new SAF for failover.
`original_mrs_ area_len`	`int32`	DL	Number of bytes in original MRS area specification for failover.
`original_mrs_ area`	256-byte array	A(256)	MRS original area specification for failover.
`original_owner_group`	`word`	DW	The current group that owns the SAF.
`new_owner_ group`	`word`	DW	The new group that will assume ownership of the SAF after failover is complete.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`MRS_SERVER`	`PAMS_MRS_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_SAF_SET`	`MSG_TYPE_MRS_SAF_SET`

See Also

- `MRS_SAF_SET_REP`

- `MRS_DQF_SET`
- `MRS_DQF_SET_REP`

MRS_SAF_SET_REP

Applications can request to open, close, or failover (redirect) a store-and-forward file (SAF) by sending an `MRS_SAF_SET` message to the MRS Server. The failover function renames a SAF file, associating it with another target queue that does not currently have a SAF associated with it. The `MRS_SAF_SET_REP` message returns the status of the request. See the Opening, Closing, and Failing Over SAF and DQF Files topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted. This message is RISC aligned.

C Message Structure

```
/*****************************************/
/* ACTION VALUES FOR MRS_SAF_SET message */
/*****************************************/
#define SAF_SET_OPEN 4
#define SAF_SET_CLOSE 5
#define SAF_SET_FAILOVER 6

/*****************************************/
/* STATUS VALUES FOR MRS_SAF_SET message */
/*****************************************/
#define JRN_SET_ERROR 0
#define JRN_SET_SUCCESS 1
#define JRN_SET_REFUSED 2

typedef struct _MRS_SAF_SET {
    int16 version;
    int16 action;
    int32 status;
    q_address original_target;
    q_address new_target;
    int32 original_mrs_area_len;
    char original_mrs_area [256];
    int16 original_owner_group;
    int16 new_owner_group;
    } MRS_SAF_SET;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number. Must be 0.
`action`	`word`	DW	4 = Open 5 = Close 6 = Failover
`status`	`int32`	DL	0 = Error 1 = Success 2 = Refused
`original_target`	`q_address`	DL	Queue address of SAF.
`new_target`	`q_address`	DL	Queue address of new SAF for failover.
`original_mrs_ area_len`	`int32`	DL	Number of bytes in original MRS area specification for failover.
`original_mrs_ area`	256-byte array	A(256)	MRS original area specification for failover.
`original_owner_group`	`word`	DW	The current group that owns the SAF.
`new_owner_ group`	`word`	DW	The new group that will assume ownership of the SAF after failover is complete.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`MRS_SERVER`	`PAMS_MRS_SERVER`
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_SAF_SET_REP`	`MSG_TYPE_MRS_SAF_SET_REP`

See Also

- `MRS_SAF_SET`

- `MRS_DQF_SET`
- `MRS_DQF_SET_REP`

MRS_SET_DLJ

Applications can request to close a dead letter journal (DLJ) file and open a new one by sending an `MRS_SET_DLJ` message to the MRS Server. Because the DLJ file cannot be simultaneously open for read and write access, an application must close the current file to read from it and open a new file to continue collecting messages. See the Managing Message Recovery Files topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is sent between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted.

C Message Structure

```
typedef struct _MRS_SET_DLJ {
    int16 version;
    int32 user_tag;
    int32 status;
    char dlj_file [64];
    } MRS_SET_DLJ;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number.
`user_tag`	`int32`	XL	User-defined tag.
`status`	`int32`	XL	0=Error 1=Success 2=Refused
`dlj_file`	64-char array	A(64)	File specification of DLJ file.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`MRS_SERVER`	`PAMS_MRS_SERVER`

Argument	Script Format	pams_get_msg Format
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_SET_DLJ`	`MSG_TYPE_MRS_SET_DLJ`

See Also

- `MRS_SET_DLJ_REP`
- `MRS_SET_PCJ`
- `MRS_SET_PCJ_REP`

MRS_SET_DLJ_REP

Applications can request to close a dead letter journal (DLJ) file and open a new one by sending a `MRS_SET_DLJ` message to the MRS Server. Because the DLJ file cannot be simultaneously open for read and write access, an application must close the current file to read from it and open a new file to continue collecting messages. The `MRS_SET_DLJ_REP` message returns the status of the request. See the Managing Message Recovery Files topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted.

C Message Structure

```
typedef struct _MRS_SET_DLJ {
    int16 version;
    int32 user_tag;
    int32 status;
    char dlj_file [64];
    } MRS_SET_DLJ;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number.
`user_tag`	`int32`	XL	User-defined tag.
`status`	`int32`	XL	0=Error 1=Success 2=Refused
`dlj_file`	64-char array	A(64)	File specification of DLJ file.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`MRS_SERVER`	`PAMS_MRS_SERVER`
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_SET_DLJ_REP`	`MSG_TYPE_MRS_SET_DLJ_REP`

See Also

- `MRS_SET_DLJ`
- `MRS_SET_PCJ`
- `MRS_SET_PCJ_REP`

MRS_SET_PCJ

Applications can request to close a postconfirmation journal (PCJ) file and open a new one by sending an `MRS_SET_PCJ` message to the MRS Server. Because the PCJ file cannot be simultaneously open for read and write access, an application must close the current file to read from it and open a new file to continue collecting messages. If default journaling is enabled, all recoverable messages are written to the PCJ file after confirmation unless the confirming process overrides the default. If default journaling is disabled, only those messages that are explicitly confirmed with `PDEL_FORCE_J` are written to the PCJ file. See the Managing Message Recovery Files topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is sent between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted.

C Message Structure

```
typedef struct _MRS_SET_PCJ {
    int16 version;
    int32 user_tag;
    int32 force_j;
    int32 status;
    char pcj_file [64];
    } MRS_SET_PCJ;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number.
`user_tag`	`int32`	XL	User-defined tag.
`force_j`	`int32`	DL	0 = Disable 1 = Enable default journaling
`status`	`int32`	XL	0=Error 1=Success 2=Refused
`pcj_file`	64-char array	A(64)	File specification of PCJ file.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`MRS_SERVER`	`PAMS_MRS_SERVER`
Source	Supplied by BEA MessageQ	Supplied by BEA MessageQ
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_SET_PCJ`	`MSG_TYPE_MRS_SET_PCJ`

See Also

- `MRS_SET_PCJ_REP`
- `MRS_SET_DLJ`
- `MRS_SET_DLJ_REP`

MRS_SET_PCJ_REP

Applications can request to close a postconfirmation journal (PCJ) and open a new one by sending an `MRS_SET_PCJ` message to the MRS Server. Because the PCJ file cannot be simultaneously open for read and write access, an application must close the current file to read from it and open a new file to continue collecting messages. The `MRS_SET_PCJ_REP` message returns the status of the request. See the Managing Message Recovery Files topic in the Using Message-Based Services section for an explanation of how to use this message. This service is available on OpenVMS systems only.

Note: The MRS Server **does not** perform endian conversion when this message is received between processes that run on systems that use different hardware data formats. The sender program must convert the message to the endian format of the target system to ensure that the message data is correctly interpreted.

C Message Structure

```
typedef struct _MRS_SET_PCJ {
    int16 version;
    int32 user_tag;
    int32 force_j;
    int32 status;
    char pcj_file [64];
    } MRS_SET_PCJ;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`word`	DW	Format version number.
`user_tag`	`int32`	XL	User-defined tag.
`force_j`	`int32`	DL	0 = Disable 1 = Enable default journaling
`status`	`int32`	XL	0=Error 1=Success 2=Refused
`pcj_file`	64-char array	A(64)	File specification of the PCJ.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`MRS_SERVER`	`PAMS_MRS_SERVER`
Class	`MRS`	`MSG_CLAS_MRS`
Type	`MRS_SET_PCJ_REP`	`MSG_TYPE_MRS_SET_PCJ_REP`

See Also

- `MRS_SET_PCJ`
- `MRS_SET_DLJ`
- `MRS_SET_DLJ_REP`

Q_UPDATE

Applications can register to receive notification when queue states change in local or remote groups by sending an ENABLE_Q_NOTIFY_REQ message. The ENABLE_Q_NOTIFY_RESP message delivers a list of all active queues and then subsequently notifies the application of attachments, detachments, and changes to active and inactive status using the Q_UPDATE message. See the Receiving Attachment Notifications topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Queue Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
#define MAX_NUMBER_Q_RECS 50

typedef struct _Q_NOTIFY_RESP {
    int32 version;
    int32 user_tag;
    int32 status_code;
    int32 last_block_flag;
    int32 number_q_recs;
    struct  {
        q_address q_num;
        q_address q_owner;
        int32  q_type;
        int32  q_active_flag;
        int32  q_attached_flag;
        int32  q_owner_pid;
        } q_rec [50];
} Q_NOTIFY_RESP;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	int32	DL	Version of response.
user_tag	int32	DL	User-specified code from request.
status_code	int32	DL	0=Error 1=Success 2=Refused

Field	Data Type	Script Format	Description
`last_block_flag`	`int32`	DL	Last block Boolean flag.
`number_q_recs`	`int32`	DL	Number of records in this message.
`q_num`	`q_address`	DL	Queue number.
`q_owner`	`q_address`	DL	Queue owner (only for secondary queues (SQs)).
`q_type`	`int32`	DL	Queue type (numerically encoded P, S, M).
`q_active_flag`	`int32`	DL	Queue active Boolean flag.
`q_attached_flag`	`int32`	DL	Queue attached Boolean flag.
`q_owner_pid`	`int32`	DL	Queue owner process identification (PID).

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`QUEUE_SERVER`	`PAMS_QUEUE_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`Q_UPDATE`	`MSG_TYPE_Q_UPDATE`

See Also

- `ENABLE_Q_NOTIFY_REQ`
- `ENABLE_Q_NOTIFY_RESP`
- `DISABLE_Q_NOTIFY_REQ`
- `DISABLE_Q_NOTIFY_RESP`

SBS_DEREGISTER_REQ

Requests SBS deregistration by exact match of MOT and distribution queue or by registration ID.

This service replaces the `SBS_DEREG` service.

Note: The SBS Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _SBS_DEREGISTER_REQ {
    int32 version;
    int32 user_tag;
    int32 mot;
    q_address distribution_q;
    int32 reg_id;
    int32 req_ack;
    } SBS_DEREGISTER_REQ;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Message format version number. Must be 40.
`user_tag`	`int32`	DL	User-specified code to identify this request.
`mot`	`int32`	DL	The MOT broadcast stream from which the program wants to deregister. 0 if unused.
`distribution_q`	`q_address`	DW, DW	The BEA MessageQ address of the distribution queue of the registration. A zero in the group number portion of the queue address automatically is replaced with the group number of the sender.
`reg_id`	`int32`	DL	The ID of the registration request to deregister. 0 if unused.

Field	Data Type	Script Format	Description
`req_ack`	`int32`	DL	1 if registration acknowledgment message is required; 0 otherwise.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`SBS_SERVER`	`PAMS_SBS_SERVER`
Source	Source queue address of the requester.	Source queue address of the requester.
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`SBS_DEREGISTER_REQ`	`MSG_TYPE_SBS_ DEREGISTER_REQ`

See Also

- `SBS_DEREGISTER_RESP`
- `SBS_REGISTER_REQ`
- `SBS_REGISTER_RESP`

SBS_DEREGISTER_RESP

This response message acknowledges the SBS server deregistration of all entries matching the given MOT queue and distribution queue.

This service replaces the `SBS_DEREG_ACK` service.

Note: The SBS Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _SBS_DEREGISTER_RESP {
    int32 version;
    int32 status;
    int32 user_tag;
    int32 number_reg;
    } SBS_DEREGISTER_RESP;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Message format version number. Must be 40.

`status`	`int32`	DL	Returned status code. Valid codes are as follows:
			PSYM_SBS_SUCCESS = Success
			PSYM_SBS_BADPARAM = Bad parameter
			PSYM_SBS_NOMATCH = No match

`user_tag`	`int32`	DL	User-specified code from the request message.
`number_reg`	`int32`	DL	The number of registrants left on this MOT or target.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program

Argument	Script Format	pams_get_msg Format
Source	`SBS_SERVER`	`PAMS_SBS_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`SBS_DEREGISTER_RESP`	`MSG_TYPE_SBS_ DEREGISTER_RESP`

See Also

- `SBS_DEREGISTER_REQ`
- `SBS_REGISTER_REQ`
- `SBS_REGISTER_RESP`

SBS_REGISTER_REQ

This request message requests registration for reception of broadcast messages. It can specify from 0 to 255 distribution rules, which must be satisfied for the message to be distributed to the distribution queue. If a sequence gap notification is requested, an SBS_SEQUENCE_GAP message is sent to the distribution queue every time a .message sequence gap is detected.

This service replaces the SBS_REG and SBS_REG_EZ services.

Note: The SBS Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _SBS_REGISTER_HEAD {
    int32 version;
    int32 user_tag;
    int32 mot;
    q_address distribution_q;
    int32 req_ack;
    int32 seq_gap_notify;
    int32 auto_dereg;
    int32 rule_count;
    int32 rule_conjunct;
    } SBS_REGISTER_HEAD;

typedef struct _SBS_REGISTER_RULE {
    int32 offset;
    int32 data_operator;
    int32 length;
    int32 operand;
    } SBS_REGISTER_RULE;

#define MAX_SEL_RULES 256
typedef struct _SBS_REGISTER_REQ {
    SBS_REGISTER_HEAD head;
    SBS_REGISTER_RULE rule [256];
    } SBS_REGISTER_REQ;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	int32	DL	Message format version number. Must be 40.
user_tag	int32	DL	User-specified code to identify this request.
mot	int32	DL	The MOT broadcast stream to which the program attempts to register.
distribution_q	q_address	DW, DW	The BEA MessageQ address that receives any messages that are selected from the broadcast stream. A zero in the group number portion of the queue address is automatically replaced with the group number of the sender.
req_ack	int32	DL	1 if registration acknowledgment message is required; 0 otherwise.
seq_gap_notify	int32	DL	1 if broadcast stream sequence gap notification is required; 0 otherwise.
auto_dereg	int32	DL	1 if registration request is to be purged on distribution queue detach; 0 otherwise.
rule_count	int32	DL	Number of distribution rules in the request (0, ..., 255).
rule_conjunct	int32	DL	Valid values are: PSEL_ALL_RULES if all rules must be true for distribution to succeed; PSEL_ANY_RULE if any rule being true can trigger distribution.
* Following items are repeated "rule_count" times *			
data_offset	int32	DL	Valid values are: PSEL_TYPE PSEL_CLAS SDM tag ID Integer in the range 0, ..., MAX_MSG_SIZE, specifying an offset in the data

Field	Data Type	Script Format	Description
`data_operator`	`int32`	DL	Valid values are: `PSEL_OPER_ANY` (always match) `PSEL_OPER_EQ` (equal) `PSEL_OPER_NEQ` (not equal) `PSEL_OPER_GTR` (greater than) `PSEL_OPER_LT` (less than) `PSEL_OPER_GTRE` (greater than or equal) `PSEL_OPER_LTE` (less than or equal) `PSEL_OPER_AND` ("operand" field AND data at "data offset" is non-zero)
`data_length`	`int32`	DL	Specifies the size of comparison to be performed: One of 0, 1, 2, or 4 bytes.
`operand`	`int32`	DL	Value used for comparison with data at the "data offset".

Arguments

Argument	Script Format	pams_get_msg Format
Target	`SBS_SERVER`	`PAMS_SBS_SERVER`
Source	Requesting program	Requesting program
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`SBS_REGISTER_REQ`	`MSG_TYPE_SBS_REGISTER_REQ`

See Also

- `SBS_DEREGISTER_REQ`
- `SBS_DEREGISTER_RESP`
- `SBS_REGISTER_RESP`
- `SBS_SEQUENCE_GAP`

SBS_REGISTER_RESP

This message provides a response to an SBS_REGISTER_REQ subscriber registration. The response contains a status field, which is 1 on success. The message also contains the user tag, specified in the request message, the registration ID and the number of registered entries for the MOT address.

This service replaces the SBS_REG_REPLY and SBS_REG_EZ_REPLY services.

Note: The SBS Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _SBS_REGISTER_RESP {
    int32 version;
    int32 user_tag;
    int32 status;
    int32 reg_id;
    int32 number_reg;
    } SBS_REGISTER_RESP;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	int32	DL	Message format version number. Must be 40.
user_tag	int32	DL	User-specified code from the request message.

status	int32	DL	Returned status code. Valid codes are as follows:
			PSYM_SBS_SUCCESS = Success
			PSYM_SBS_BADPARAM = Bad parameter
			PSYM_SBS_RESRCFAIL = Failed to allocate resource

reg_id	int32	DL	Returned registration ID.
number_reg	int32	DL	Number of entries currently registered for this MOT or target.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Source of registrant	Source of registrant
Source	`SBS_SERVER`	`PAMS_SBS_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`SBS_SEQUENCE_RESP`	`MSG_TYPE_SBS_SEQUENCE_ RESP`

See Also

- `SBS_DEREGISTER_REQ`
- `SBS_DEREGISTER_RESP`
- `SBS_REGISTER_REQ`

SBS_SEQUENCE_GAP

This message indicates that a sequence gap occurred in a broadcast stream. Sequence gaps can occur when the sender is broadcasting at a higher rate than the receiver can handle.

This service replaces the `SBS_BS_SEQGAP` service.

Note: The SBS Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _SBS_SEQUENCE_GAP {
    int32 num_msgs_missing;
    int32 sender_group;
    int32 mot;
    int32 channel;
    } SBS_SEQUENCE_GAP;
```

Message Data Fields

Field	Data Type	Script Format	Description
`num_msgs_ missing`	`int32`	DL	Number of lost messages in sequence gap.
`sender_group`	`int32`	DL	Group number of sending SBS server.
`mot`	`int32`	DL	MOT address in which broadcast stream gap occurred.
`channel`	`in32`	DL	Source address of MOT; either SBS server or Ethernet channel.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Registrant	Registrant
Source	`SBS_SERVER`	`PAMS_SBS_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`

Argument	Script Format	pams_get_msg Format
Type	`SBS_SEQUENCE_GAP`	`MSG_TYPE_SBS_SEQUENCE_ GAP`

See Also

- `SBS_REGISTER_REQ`

SBS_STATUS_REQ

The SBS server supports a message-based status request. This request details the current condition of each MOT being used by the server and its activity with other BEA MessageQ groups, which are also running the SBS server.

The request message is targeted to the `SBS_SERVER` with message class PAMS and message type `SBS_STATUS_REQ`. Upon receipt of the message, the SBS server validates the request. If the request is incorrect, the response message contains an error status. The SBS server responds with the reply message of type `SBS_STATUS_RESP`.

Note: The SBS Server performs endian conversion when this message is sent between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _SBS_STATUS_REQ {
    int32 version;
    int32 user_tag;
    int32 start_mot;
    int32 end_mot;
    int32 reset;
    } SBS_STATUS_REQ;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Message format version number. Must be 40.
`user_tag`	`int32`	DL	User-specified code to identify this request.
`start_mot`	`int32`	DL	Lowest MOT for which statistics are desired.
`end_mot`	`int32`	DL	Highest MOT for which statistics are desired.
`reset`	`int32`	DL	0: Do not reset counters for the remote server data after constructing the reply message. 1: Reset counters for the remote server data after constructing the reply message.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`SBS_SERVER`	`PAMS_SBS_SERVER`
Source	Requesting program’s primary or reply queue	Requesting program’s primary or reply queue
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`SBS_STATUS_REQ`	`MSG_TYPE_SBS_STATUS_REQ`

See Also

- `SBS_STATUS_RESP`

SBS_STATUS_RESP

This message is returned following the successful processing of the `SBS_STATUS_REQ` request message. It is a variable format message and is made up of a variable number of fixed length parts. To parse the message, each variable length section has a count.

Note: The SBS Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _SBS_STATUS_RESP {
    int32 version;
    int32 user_tag;
    int32 status;
    int32 num_rec;
    int32 last_block;
    char data [31980];
    } SBS_STATUS_RESP;

typedef struct _SBS_STATUS_RESP_MOT {
    int32 mot;
    union  {
        struct  {
            union   {
                 struct  {
                     char s_b1;
                     char s_b2;
                     char s_b3;
                     char s_b4;
                     } S_un_b;
                 struct  {
                     uint16 s_w1;
                     uint16 s_w2;
                     } S_un_w;
                 uint32 S_addr;
                 } inet_addr;
            uint16 inet_port;
            } udp;
        struct  {
            char mca_addr [12];
            char protocol [4];
            } eth;
        struct {
            char unused [20];
            } dmq;
        int32 filler [5];
        } transport;
    int32 heartbeat_timer;
    int32 xmit_silo;
    int32 rcv_silo;
    int32 rcv_silo_max;
    int32 num_reg;
    int32 complete_rcvd;
    int32 complete_bytes;
    int32 seq_gaps;
    int32 whole_msg_gaps;
    int32 whole_silo_gap;
    struct  {
        char device_name [16];
        struct  {
            uint32 tv_sec;
            uint32 tv_usec;
            } fail_tod;
        int32 msgs_sent;
        int32 bytes_sent;
        int32 pkts_sent;
        int32 pkts_rcvd;
        int32 dupl_pkts_disc;
        } rail [2];
    } SBS_STATUS_RESP_MOT;

typedef struct _SBS_STATUS_REP_REG_Q {
    q_address reg_q;
    } SBS_STATUS_REP_REG_Q;

typedef struct _SBS_STATUS_REP_NUM_GROUPS {
    int32 num_groups;
    } SBS_STATUS_REP_NUM_GROUPS;

typedef struct _SBS_STATUS_RESP_GROUP {
    int32 group;
    int32 rexmit_reqs_to_remote;
    int32 rexmit_sat_by_remote;
    int32 late_rexmit;
    int32 rexmit_reqs_from_remote;
    int32 rexmit_sat_by_local;
    } SBS_STATUS_RESP_GROUP;
```

Message Data Fields

Field	Data Type	Script Format	Description
`version`	`int32`	DL	Message format version number. Must be 40.
`user_tag`	`int32`	DL	User-specified code to identify this request.
`Status`	`int32`	DL	Returned status code. Valid codes are as follows: `PSYM_SBS_SUCCESS` = Success `PSYM_SBS_BADPARAM` = Bad parameter `PSYM_SBS_NOMATCH` = No match
`num_rec`	`int32`	DL	Number of MOTs reported in this message.
`last_block`	`int32`	DL	1 if this is the last message; 0 otherwise.
* Remainder of message repeated "`num_rec`" times up to a maximum of 50 records per Local SBS Server data *			
`mot`	`int32`	DL	MOT for which statistics are being reported.
`transport`		A(20)	Transport specific address information associated with the MOT. The format is dependant on the type of transport referred to.
`heartbeat_timer`	`int32`	DL	Heartbeat timer setting.
`xmit_silo`	`int32`	DL	Transmit silo size (MABs).
rcv_silo	int32	DL	Receiver silo size (MABs).
`rcv_silo_max`	`int32`	DL	Maximum occupancy of receive silo (MABs).
`num_reg`	`int32`	DL	Number of registrants for this MOT.
`complete_rcvd`	`int32`	DL	Number of complete messages received.
`complete_bytes`	`int32`	DL	Number bytes contained in “`complete_rcvd`” messages.

Field	Data Type	Script Format	Description
`seq_gaps`	`int32`	DL	Total sequence gaps reported on this MOT.
`whole_msg_gaps`	`int32`	DL	Number complete messages detected missed initially.
`whole_silo_gap`	`int32`	DL	Number times sequence gap caused entire silo flush.
* Transport rail information repeated two times *			
`device_name`	`char`	A(16)	Optimized device address.
`fail_tod`		DL(2)	Shutdown timestamp in seconds.
`msgs_sent`	`int32`	DL	Number of messages sent on this rail.
`bytes_sent`	`int32`	DL	Number of bytes sent on this rail.
`pkts_sent`	`int32`	DL	Number of packets sent on this rail.
`pkts_rcvd`	`int32`	DL	Number of packets received on rail.
`dupl_pkts_disc`	`int32`	DL	Number of duplicate packets discarded from this rail.
* Registrant data: repeated "`num_reg`" times *			
`reg_q`	`q_address`	DW, DW	Queue address of registrant.
* End of registrant data *			
`num_groups`	`int32`	DL	Number of remote SBS servers communicating with the local SBS server.
* Remote SBS server data: Following fields repeated "`num_groups`" times *			
`group`	`int32`	DL	Group number of remote SBS server.
`rexmit_reqs_to_ remote`	`int32`	DL	Number of retransmission requests from the local SBS server to the remote SBS server.
`rexmit_sat_by_ remote`	`int32`	DL	Number of retransmission requests satisfied by the remote SBS server.

Field	Data Type	Script Format	Description
`late_rexmit`	`int32`	DL	Number of retransmission requests that were received too late to prevent a sequence gap.
`rexmit_reqs_ from_remote`	`int32`	DL	Number of retransmission requests from the remote SBS server.
`rexmit_sat_by _local`	`int32`	DL	Number of retransmission requests satisfied by the local SBS server for the remote server.
* End of remote server data *			

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program's primary or reply queue	Requesting program's primary or reply queue
Source	`SBS_SERVER`	`PAMS_SBS_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`SBS_STATUS_RESP`	`MSG_TYPE_SBS_STATUS_RESP`

See Also

- `SBS_STATUS_REQ`

TIMER_EXPIRED

TIMER_EXPIRED is a response message to the pams_set_timer function. This message is sent to the timer queue associated with sender program's primary queue. Each call to the pams_set_timer function generates one message of type TIMER_EXPIRED when the timer expires.

Note: This message is RISC aligned.

C Message Structure

```
typedef struct _TIMER_EXPIRED {
    int32 timer_id;
    char reserved [20];
    } TIMER_EXPIRED;
```

Message Data Fields

Field	Data Type	Script Format	Description
timer_id	int32	DL	Timer ID specified in the pams_set_timer call.
reserved	20-byte array	A(20)	Reserved for BEA MessageQ.

Arguments

Argument	Script Format	pams_get_msg Format
Target	primary queue	primary queue
Source	TIMER_QUEUE	PAMS_TIMER_QUEUE
Class	PAMS	MSG_CLAS_PAMS
Type	TIMER_EXPIRED	MSG_TYPE_TIMER_EXPIRED

UNAVAIL

Applications register to receive notification when queues become active or inactive in local and remote groups by sending an `AVAIL_REG` message to the Avail Server. The `UNAVAIL` notification message is sent to the registered application when a queue in the selected group becomes inactive. See the Obtaining the Status of a Queue topic in the Using Message-Based Services section for an explanation of how to use this message.

Note: The Avail Server performs endian conversion when this message is received between processes that run on systems that use different hardware data formats. This message is also RISC aligned.

C Message Structure

```
typedef struct _UNAVAIL {
    q_address target_q;
    } UNAVAIL;
```

Message Data Fields

Field	Data Type	Script Format	Description
`target_q`	`q_address`	DL	Address of unavailable queue.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Supplied by `AVAIL_REG`	Supplied by `AVAIL_REG`
Source	`AVAIL_SERVER`	`AVAIL_SERVER`
Class	`PAMS`	`MSG_CLAS_PAMS`
Type	`UNAVAIL`	`MSG_TYPE_UNAVAIL`

See Also

- `AVAIL_REG`
- `AVAIL_REG_REPLY`
- `AVAIL`
- `AVAIL_DEREG`

A Supported Delivery Modes and Undeliverable Message Actions

This appendix describes the valid combinations for the `delivery` and `uma` arguments in each BEA MessageQ supported environment.

The `delivery` argument uses the `PDEL_MODE_`*`sn_dip`* format where:

- *`sn`* is one of the following sender notification codes:

  `WF`—Wait for completion

  `AK`—Asynchronous acknowledgment

  `NN`—No notification

- *`dip`* is one of the following delivery interest point codes:

  `ACK`—Read from target queue and explicitly acknowledged using the `pams_confirm_msg` service. `ACK` can also be an implicit acknowledgement sent after the second `pams_get_msg` call by the receiving application.

  `CONF`—Delivered from the DQF and explicitly confirmed using the `pams_confirm_msg` service

  `DEQ`—Read from the target queue

  `DQF`—Stored in the destination queue file

`MEM`—Stored in the target queue

`SAF`—Stored in the store and forward file

Note: If temporary queues are used, deleted, and reused quickly, it is possible in isolated cases for an implicit ACK response from a previous temporary queue to be placed on the new temporary queue.

The `uma` argument uses the `PDEL_UMA_`*`xxx`* format where *`xxx`* is one of the following codes:

`DISC`—Discard

`DISCL`—Discard after logging (Open VMS only)

`DLJ`—Dead letter journal

`DLQ`—Dead letter queue

`RTS`—Return to sender

`SAF`—Store and forward

Note: On UNIX and Windows NT systems, the `DISCL` UMA performs the same as the `DISC` UMA, discarding the message without logging the event.

Delivery Mode and UMA Cross-Reference

Table A-1 uses the following key for delivery mode support:

X—-Supported
`.`—-Not supported
S—-Available if supported by message ser

Table A-1 Delivery Mode and UMA Cross-Reference

Delivery Mode Version	UMA	UNIX/ Windows NT	OpenVMS	Clients
`PDEL_MODE_AK_ACK`	`DISC`	X	X	X
	`DISCL`	X	X	X
	`DLQ`	X	X	X
	`DLJ`	.	X	S
	`RTS`	X	X	X
	`SAF`	.	.	.
`PDEL_MODE_AK_CONF`	`DISC`	X	X	X
	`DISCL`	X	X	X
	`DLQ`	X	X	X
	`DLJ`	X	X	S
	`RTS`	X	X	X
	`SAF`	X	.	X
`PDEL_MODE_AK_DEQ`	`DISC`	X	X	X
	`DISCL`	X	X	X
	`DLQ`	X	X	X

Table A-1 Delivery Mode and UMA Cross-Reference

Delivery Mode Version	UMA	UNIX/ Windows NT	OpenVMS	Clients
	DLJ	.	X	S
	RTS	X	X	X
	SAF	.	.	.
PDEL_MODE_AK_DQF	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	X	X	X
	RTS	X	X	X
	SAF	X	X	X
PDEL_MODE_AK_MEM	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	.	X	S
	RTS	X	X	X
	SAF	.	.	.
PDEL_MODE_AK_SAF	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	X	X	X
	RTS	X	X	X
	SAF	.	X	S

Table A-1 Delivery Mode and UMA Cross-Reference

Delivery Mode Version	UMA	UNIX/ Windows NT	OpenVMS	Clients
PDEL_MODE_NN_DQF	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	X	X	X
	RTS	X	X	X
	SAF	X	X	X
PDEL_MODE_NN_MEM	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	.	X	S
	RTS	X	X	X
	SAF	.	.	.
PDEL_MODE_NN_SAF	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	X	X	X
	RTS	X	X	X
	SAF	.	X	S
PDEL_MODE_WF_ACK	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X

Table A-1 Delivery Mode and UMA Cross-Reference

Delivery Mode Version	UMA	UNIX/ Windows NT	OpenVMS	Clients
	DLJ	.	X	S
	RTS	X	X	X
	SAF	.	.	.
PDEL_MODE_WF_CONF	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	X	X	X
	RTS	X	X	X
	SAF	X	.	X
PDEL_MODE_WF_DEQ	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	.	X	S
	RTS	X	X	X
	SAF	.	.	.
PDEL_MODE_WF_DQF	DISC	X	X	X
	DISCL	X	X	X
	DLQ	X	X	X
	DLJ	X	X	X
	RTS	X	X	X
	SAF	X	X	X

Table A-1 Delivery Mode and UMA Cross-Reference

Delivery Mode Version	UMA	UNIX/ Windows NT	OpenVMS	Clients
`PDEL_MODE_WF_MEM`	`DISC`	X	X	X
	`DISCL`	X	X	X
	`DLQ`	X	X	X
	`DLJ`	.	X	S
	`RTS`	X	X	X
	`SAF`	.	.	X
`PDEL_MODE_WF_SAF`	`DISC`	X	X	X
	`DISCL`	X	X	X
	`DLQ`	X	X	X
	`DLJ`	X	X	X
	`RTS`	X	X	X
	`SAF`	.	X	S

Key to Delivery Modes Supported

X—-Supported
.—-Not supported
S—-Available if supported by message server

B Obsolete Functions and Services

This appendix contains reference information for obsolete functions and services. These functions and services should not be used in new development. Information is provided referencing features which replace obsolete functions and services.

Obsolete Message-Based Services for Message Broadcasting

This section contains reference information for the following obsolete services for message broadcasting:

- `SBS_BS_SEQGAP`
- `SBS_DEREG`
- `SBS_DEREG_ACK`
- `SBS_DEREG_BY_ID`
- `SBS_REG`
- `SBS_REG_EZ`
- `SBS_REG_EZ_REPLY`
- `SBS_REG_REPLY`

SBS_BS_SEQGAP

Note: This service is obsolete. Use SBS_SEQUENCE_GAP instead.

Applications can register to receive notification of sequence gaps in broadcast messages when sending the SBS_REG message to the SBS Server. The registered application receives an SBS_BS_SEQGAP message when there is a gap in sequence of broadcast messages. Sequence gaps can occur when the sender program is broadcasting at a higher rate than the receiver program can handle.

C Message Structure

```
typedef struct _SBS_BS_SEQGAP {
    int32 num_msgs_missing;
    uint16 sender_group;
    uint16 mot;
    uint16 channel;
    } SBS_BS_SEQGAP;
```

Message Data Fields

Field	Data Type	Script Format	Description
num_msgs_ missing	int32	DL	Count of lost messages in sequence gap.
sender_group	unsigned word	DW	Group address of sending SBS Server.
mot	unsigned word	DW	Multipoint Outbound Target (MOT) address in which broadcast stream gap occurred.
channel	unsigned word	DW	Source address of MOT; either SBS Server or Ethernet channel.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	SBS_SERVER	PAMS_SBS_SERVER
Class	PAMS	MSG_CLAS_PAMS

Argument	Script Format	pams_get_msg Format
Type	`SBS_BS_SEQGAP`	`MSG_TYPE_SBS_BS_SEQGAP`

See Also

- `SBS_REG`
- `SBS_DEREG`

SBS_DEREG

Note: This service is obsolete. Use SBS_DEREGISTER_REQ instead.

Applications can register to receive broadcast messages by sending an SBS_REG message or an SBS_REG_EZ message to the SBS Server. When an application no longer needs to receive messages from a broadcast stream, it sends an SBS_DEREG message to the SBS Server. This message causes the SBS Server to deregister all entries for the broadcast stream and receiving queue combination.

C Message Structure

```
typedef struct _SBS_DEREG {
    int16 version;
    uint16 mot;
    q_address distribution_q;
    char req_ack;
    } SBS_DEREG;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	word	DW	Message format version. Must be 20.
mot_q	unsigned word	DW	MOT queue address.
distribution_q	q_address	DL	Distribution queue address.
req_ack	Boolean	DB	Value of 1 if acknowledgment requested.

Arguments

Argument	Script Format	pams_get_msg Format
Target	SBS_SERVER	PAMS_SBS_SERVER
Source	Requesting program	Requesting program
Class	PAMS	MSG_CLAS_PAMS
Type	SBS_DEREG	MSG_TYPE_SBS_DEREG

See Also

- `SBS_BS_SEQGAP`
- `SBS_DEREG_ACK`
- `SBS_DEREG_BY_ID`
- `SBS_REG`
- `SBS_REG_EZ`

SBS_DEREG_ACK

Note: This service is obsolete. Use `SBS_DEREGISTER_RESP` instead.

Applications can register to receive broadcast messages by sending an `SBS_REG` message or an `SBS_REG_EZ` message to the SBS Server. When an application no longer needs to receive messages from a broadcast stream, it sends an `SBS_DEREG` message to the SBS Server. This message causes the SBS Server to deregister all entries for the broadcast stream and receiving queue combination. The `SBS_DEREG_ACK` message acknowledges deregistration for the broadcast stream and receiver queue selected.

C Message Structure

```
typedef struct _SBS_DEREG_ACK {
    int16 status;
    int16 number_reg;
    } SBS_DEREG_ACK;
```

Message Data Fields

Field	Data Type	Script Format	Description
`status`	word	DW	The return status of 1 = success, –*n* = failure.
`number_reg`	word	DW	Number of registrants left on this Multipoint Outbound Target (MOT) after deregistration.

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	`SBS_SERVER`	`PAMS_SBS_SERVER`
Class	PAMS	`MSG_CLAS_PAMS`
Type	`SBS_DEREG_ACK`	`MSG_TYPE_SBS_DEREG_ACK`

See Also

- `SBS_DEREG`
- `SBS_DEREG_BY_ID`
- `SBS_REG`

- `SBS_REG_EZ`

SBS_DEREG_BY_ID

Note: This service is obsolete. Use SBS_DEREGISTER_REQ instead.

Applications can register to receive broadcast messages by sending an SBS_REG message or an SBS_REG_EZ message to the SBS Server. When an application has multiple registrations for a broadcast stream and no longer needs to receive one type of message, the application can send an SBS_DEREG_BY_ID message to the SBS Server by providing the ID returned by MessageQ during the initial broadcast registration. The queue will continue to receive broadcast messages requested through separate registrations.

C Message Structure

```
typedef struct _SBS_DEREG_BY_ID {
    short version;
    unsigned short reg_id;
    char req_ack;
    } SBS_DEREG_BY_ID;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	word	DW	Message format version. Must be 20.
reg_id	unsigned word	DW	Registration ID.
req_ack	Boolean	DB	Value of 1 if ACK requested.

Arguments

Argument	Script Format	pams_get_msg Format
Target	SBS_SERVER	PAMS_SBS_SERVER
Source	Requesting program	Requesting program
Class	PAMS	MSG_CLAS_PAMS
Type	SBS_DEREG_BY_ID	MSG_TYPE_SBS_DEREG_BY_ID

See Also

- SBS_DEREG

- `SBS_DEREG_ACK`
- `SBS_REG`
- `SBS_REG_EZ`

SBS_REG

Note: This service is obsolete. Use SBS_REGISTER_REQ instead.

Applications can register to receive selected messages from a broadcast stream by sending an SBS_REG message to the SBS Server. This message requests a target queue to receive all messages that meet the selection criteria entered as part of the registration process. Selection rules define a relational operation to be applied against a message header or message data field. Each broadcast message that matches the rule is distributed to the target queue.

C Message Structure

```
typedef struct _SBS_REG {
    int16 version;
    uint16 mot;
    q_address distribution_q;
    int16 offset;
    char data_operator;
    int16 length;
    uint32 operand;
    char req_ack;
    char req_seqgap;
    char req_autodereg;
    } SBS_REG;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	word	DW	Message format version number. Must be 20.
mot_addr	unsigned word	DW	The Multipoint Outbound Target (MOT) broadcast stream to which the program tries to register.
distribution_q	q_address	DL	The MessageQ address that receives any messages that are selected from the broadcast stream.
offset	word	DW	Specifies a field in the message header or in the message data component.

Field	Data Type	Script Format	Description
`operator`	byte	DB	Controls the type of comparison to be performed on the field designated by the data offset and the operand.
`length`	word	DW	Specifies the size of comparison to be performed. The choices are 0, 1, 2, and 4.
`operand`	uint32	DL	The value to be used in the comparison of the field specified by the data offset.
`req_ack`	Boolean	DB	Specifies if an acknowledgment message is requested. See `SBS_REG_REPLY`.
`req_seqgap`	Boolean	DB	Specifies if a notification of broadcast stream message sequence number gap is requested.
`req_autodereg`	Boolean	DB	Specifies if a registration request is to be automatically purged from the SBS Server table.

Arguments

Argument	Script Format	pams_get_msg Format
Target	`SBS_SERVER`	`PAMS_SBS_SERVER`
Source	Requesting program	Requesting program
Class	PAMS	`MSG_CLAS_PAMS`
Type	`SBS_REG`	`MSG_TYPE_SBS_REG`

See Also

- `SBS_REG_REPLY`
- `SBS_REG_EZ`
- `SBS_DEREG`

SBS_REG_EZ

Note: This service is obsolete. Use SBS_REGISTER_REQ instead.

Applications can register to receive all messages from a broadcast stream by sending an SBS_REG_EZ message to the SBS Server. This message requests a target queue to receive all messages sent to the selected broadcast stream.

C Message Structure

```
typedef struct _SBS_REG_EZ {
    int16 version;
    int16 mot;
    q_address distribution_q;
    } SBS_REG_EZ;
```

Message Data Fields

Field	Data Type	Script Format	Description
version	word	DW	Message format version number. Must be 20.
mot_addr	word	DW	The Multipoint Outbound Target (MOT) broadcast stream to which the process tries to subscribe.
distribution_q	q_address	DL	The MessageQ address that receives any messages selected from the broadcast stream.

Arguments

Argument	Script Format	pams_get_msg Format
Target	SBS_SERVER	PAMS_SBS_SERVER
Source	Requesting program	Requesting program
Class	PAMS	MSG_CLAS_PAMS
Type	SBS_REG_EZ	MSG_TYPE_SBS_REG_EZ

See Also

- SBS_REG_EZ_REPLY

- `SBS_REG`
- `SBS_DEREG`

SBS_REG_EZ_REPLY

Note: This service is obsolete. Use SBS_REGISTER_RESP instead.

Applications can register to receive all messages from a broadcast stream by sending an SBS_REG_EZ message to the SBS Server. This message requests that all messages sent to a broadcast stream be distributed to a particular target queue. The SBS_REG_EZ_REPLY message indicates the status of the request and returns a registration ID if the application is successfully registered.

C Message Structure

```
typedef struct _SBS_REG_EZ_REPLY {
    int16 status;
    uint16 reg_id;
    int16 number_reg;
    } SBS_REG_EZ_REPLY;
```

Message Data Fields

Field	Data Type	Script Format	Description
status	word	DW	The return status of 1indicates success; –*n* indicates failure.
reg_id	unsigned word	DW	Returned registration ID.
number_reg	word	DW	Number of registrants left on this Multipoint Outbound Target (MOT).

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	SBS_SERVER	PAMS_SBS_SERVER
Class	PAMS	MSG_CLAS_PAMS
Type	SBS_REG_EZ_REPLY	MSG_TYPE_SBS_REG_EZ_REPLY

See Also

- SBS_REG_EZ

- `SBS_DEREG`

SBS_REG_REPLY

Note: This service is obsolete. Use SBS_REGISTER_RESP instead.

Applications can register to receive selected messages from a broadcast stream by sending an SBS_REG message to the SBS Server. This message requests a target queue to receive all messages sent to a particular broadcast stream that meet selection criteria. The SBS_REG_REPLY message indicates the status of the request and returns a registration ID.

C Message Structure

```
typedef struct _SBS_REG_REPLY {
    int16 status;
    uint16 reg_id;
    int16 number_reg;
    } SBS_REG_REPLY;
```

Message Data Fields

Field	Data Type	Script Format	Description
status	word	DW	The return status of 1 = success, –*n* = failure.
reg_id	unsigned word	DW	Returned registration ID.
number_reg	word	DW	Number of registrants left on this Multipoint Outbound Target (MOT).

Arguments

Argument	Script Format	pams_get_msg Format
Target	Requesting program	Requesting program
Source	SBS_SERVER	PAMS_SBS_SERVER
Class	PAMS	MSG_CLAS_PAMS
Type	SBS_REG_REPLY	MSG_TYPE_SBS_REG_REPLY

See Also

- SBS_REG

Obsolete PAMS API Functions

This section contains reference information for the following obsolete PAMS API functions:

- `pams_create_handle`
- `pams_decode`
- `pams_delete_handle`
- `pams_encode`
- `pams_extract_buffer`
- `pams_insert_buffer`
- `pams_msg_length`
- `pams_next_msg_field`
- `pams_remove_encoding`
- `pams_set_msg_position`

pams_create_handle

Note: This function is obsolete. Handles and the SDM format used in MessageQ Version 4.0 have been replaced by the FML32 format for self describing messaging. Use `pams_put_msg` with the `PSYM_MSG_FML` symbol and `pams_get_msg(w)` with the `PSYM_MSG_BUFFER_PTR` symbol to send and receive FML messages.

Creates an empty message and returns a handle to it.

Syntax

```
int32 pams_create_handle ( handle, [ handle_type ] );
```

Arguments

Table 9-1

Argument	Data Type	Mechanism	Prototype	Access
`handle`	`pams_handle`	reference	`pams_handle` *	returned
`handle_type`	int32	reference	int32 *	passed

Argument Definitions

handle

Supplies the handle that you want created.

handle_type

Specifies the type of handle to create.

Return Values

Table 9-2

Return Code	Platform	Description
`PAMS__BADARGLIST`	All	Invalid number of arguments.
`PAMS__BADPARAM`	All	Invalid `handle_type` argument value.
`PAMS__RESRCFAIL`	All	Insufficient resources to complete the operation.
`PAMS__SUCCESS`	All	Indicates successful completion.

Description

The **`handle_type`** argument takes the `PSYM_MSG_HANDLE` value to request the creation of a message handle. In this case, the returned `pams_handle` argument can be used everywhere a **`pams_handle`** data type is needed with a function.

PSYM_MSG_HANDLE is a default value, so providing a null pointer as `handle_type` creates a message handle.

See Also

- `pams_delete_handle`
- `pams_extract_buffer`
- `pams_insert_buffer`
- `pams_msg_length`

pams_decode

Note: This function is obsolete. Handles and the SDM format used in MessageQ Version 4.0 have been replaced by the FML32 format for self describing messaging. Use `pams_put_msg` with the `PSYM_MSG_FML` symbol and `pams_get_msg(w)` with the `PSYM_MSG_BUFFER_PTR` symbol to send and receive FML messages.

The pams_decode functions are a series of functions that decode a tagged field out of the message. The first unseen field in the message with the desired element is returned. The actual name of the function and its description is listed as follows:

Table 9-3

Function	Description
`pams_decode_int8`	Decodes an 8-bit signed integer (char) element of information out of the message.
`pams_decode_uint8`	Decodes an 8-bit unsigned integer (unsigned char) element of information out of the message.
`pams_decode_int16`	Decodes a 16-bit signed integer element of information out of the message.
`pams_decode_uint16`	Decodes a 16-bit unsigned integer element of information out of the message.
`pams_decode_int32`	Decodes a 32-bit signed integer element of information out of the message.
`pams_decode_uint32`	Decodes a 32-bit unsigned integer element of information out of the message.
`pams_decode_int64`	Decodes a 64-bit signed integer element of information out of the message.
`pams_decode_uint64`	Decodes a 64-bit unsigned integer element of information out of the message.
`pams_decode_float`	Decodes a single floating-point element of information out of the message.
`pams_decode_double`	Decodes a double floating-point element of information out of the message.

Table 9-3

Function	Description
`pams_decode_string`	Decodes a string element of information out of the message.
`pams_decode_array`	Decodes an array of elements of information of the same type out of the message.
`pams_decode_qid`	Decodes the q_address (MessageQ queue address) out of the message.

Syntax

The syntax for each of the pams_decode functions is as follows:

Listing 9-1 Syntax for pams_encode function

```
int32 pams_decode_int8 ( pams_handle handle, int32* tag, int8*
                     value);

int32 pams_decode_uint8 ( pams_handle handle, int32* tag, uint8*
                     value);

int32 pams_decode_int16 ( pams_handle handle, int32* tag, int16*
                     value);

int32 pams_decode_uint16 ( pams_handle handle, int32* tag, uint16*
                     value);

int32 pams_decode_int32 ( pams_handle handle, int32* tag, int32*
                     value);

int32 pams_decode_uint32 ( pams_handle handle, int32* tag, uint32*
                     value);

int32 pams_decode_int64 ( pams_handle handle, int32* tag, int64*
                     value);

int32 pams_decode_uint64 ( pams_handle handle, int32* tag, uint64*
                     value);

int32 pams_decode_float ( pams_handle handle, int32* tag, float*
                     value);

int32 pams_decode_double ( pams_handle handle, int32* tag, double*
                     value);
```

```
int32 pams_decode_string ( pams_handle handle, int32* tag, char*
                     value,
                     int32* bufferLength,
                        int32* valueLength);

int32 pams_decode_array ( pams_handle handle, int32* tag, void*
                     value,
                     int32* bufferLength,
                        int32* numEltsValue);

int32 pams_decode_qid ( pams_handle handle, int32* tag,
                     q_address* value);,
```

Arguments

The following table describes the arguments for the pams_decode functions above. Some of these arguments apply to certain functions only.

Table 9-4

Argument	Data Type	Mechanism	Prototype	Access
`handle`	pams_handle	reference	`pams_handle` *	passed
`tag`	int32	reference	int32 *	passed
`value`	varying pointer	reference	int32 *	returned
`bufferLength`	int32	reference	int32 *	passed
`valueLength`	int32	reference	int32 *	returned
`numEltsValue`	int32	reference	int32 *	returned

Argument Definitions

handle

Specifies the message handle.

tag

Specifies the tag of the field to decode.

value

Contains the pointer to a buffer to receive the value of the field to decode.

bufferLength

Contains the number of bytes in the value buffer.

valueLength

Specifies the number of bytes in the returned value, unless a NULL pointer is passed. If the `valueLength` argument is a NULL pointer to the call to pams_decode_string, the string is returned null-terminated. In this case, the specified `bufferLength` argument must include space for the trailing null.

numEltsValue

Specifies the number of elements contained in the array.

Return Values

Table 9-5

Return Code	Platform	Description
`PAMS__AREATOSMALL`	All	The buffer length is too small to fit the value string.
`PAMS__BADHANDLE`	All	Invalid message handle or handle to an untyped message.
`PAMS__BADTAG`	All	The tag data type does not match the routine used for the value data type.
`PAMS__SUCCESS`	All	Indicates successful completion.
`PAMS__TAGNOTFOUND`	All	Tag not found in the message.

Description

This function scans the message for an unseen instance of the specified tag, starting at the beginning of the message:

- If the scan finds a match, the matched element is marked as seen, its value (and length) are returned, and the return value is set to `PAMS__SUCCESS`.
- If the scan continues without success to the end of the message, the return value is set to `PAMS__NOSUCHTAG`.

Thus, if there are two occurrences of a particular tag in a message, the first decode call always returns the earlier occurrence of the tag, and the second call returns the later one. Conversely, if a receiving application knows a message's tag order is Atag, Btag, ENDtag, Atag, Btag, ENDtag, the application cannot skip the first pair by decoding first ENDtag, and then decoding Atag.

The value is returned in the local host representation (endian conversions are applied when necessary).

See Also

- `pams_encode`
- `pams_remove_encoding`

pams_delete_handle

Note: This function is obsolete. Handles and the SDM format used in MessageQ Version 4.0 have been replaced by the FML32 format for self describing messaging. Use `pams_put_msg` with the `PSYM_MSG_FML` symbol and `pams_get_msg(w)` with the `PSYM_MSG_BUFFER_PTR` symbol to send and receive FML messages.

Releases all of the resources allocated for the message handle.

Syntax

```
int32 pams_delete_handle ( handle );
```

Arguments

Table 9-6

Argument	Data Type	Mechanism	Prototype	Access
`handle`	`pams_handle`	reference	`pams_handle *`	passed

Argument Definition

handle

Specifies the message handle to delete.

Return Values

Table 9-7

Return Code	Platform	Description
`PAMS__BADARGLIST`	All	Invalid number of arguments.
`PAMS__BADHANDLE`	All	Invalid message handle.
`PAMS__SUCCESS`	All	Indicates successful completion.

Description

Applications must use this function after sending or receiving messages. The use of this function avoids memory leaks. Note that your application must call this function if you decide not to send a message you have created.

See Also

- `pams_create_handle`
- `pams_extract_buffer`
- `pams_insert_buffer`

- `pams_msg_length`

pams_encode

Note: This function is obsolete. Handles and the SDM format used in MessageQ Version 4.0 have been replaced by the FML32 format for self describing messaging. Use `pams_put_msg` with the `PSYM_MSG_FML` symbol and `pams_get_msg(w)` with the `PSYM_MSG_BUFFER_PTR` symbol to send and receive FML messages.

The `pams_encode` functions are a series of functions that append a field in the SDM message based on a specific data type. The actual name of the function and its description is as follows:

Function	**Description**
`pams_encode_int8`	Encodes an 8-bit signed integer (char) element of information in the message.
`pams_encode_uint8`	Encodes an 8-bit unsigned integer (unsigned char) element of information in the message.
`pams_encode_int16`	Encodes a 16-bit signed integer element of information in the message.
`pams_encode_uint16`	Encodes a 16-bit unsigned integer element of information in the message.
`pams_encode_int32`	Encodes a 32-bit signed integer element of information in the message.
`pams_encode_uint32`	Encodes a 32-bit unsigned integer element of information in the message.
`pams_encode_int64`	Encodes a 64-bit signed integer element of information in the message.
`pams_encode_uint64`	Encodes a 64-bit unsigned integer element of information in the message.
`pams_encode_float`	Encodes a single floating-point element of information in the message.
`pams_encode_double`	Encodes a double floating-point element of information in the message.

Function	Description
`pams_encode_string`	Encodes a string element of information in the message.
`pams_encode_array`	Encodes an array of elements of information of the same type in the message.
`pams_encode_qid`	Encodes the q_address (MessageQ queue address) in the message.

Syntax The syntax for each of the `pams_encode` functions is as follows:

Listing 9-2 Syntax for pams_encode_functions

```
int32 pams_encode_int8 ( pams_handle handle, int32* tag, int8*
                      value);

int32 pams_encode_uint8 ( pams_handle handle, int32* tag, uint8*
                      value);

int32 pams_encode_int16 ( pams_handle handle, int32* tag, int16*
                      value);

int32 pams_encode_uint16 ( pams_handle handle, int32* tag,
                      uint16* value);

int32 pams_encode_int32 ( pams_handle handle, int32* tag, int32*
                      value);

int32 pams_encode_uint32 ( pams_handle handle, int32* tag, uint32*
                      value);

int32 pams_encode_int64 ( pams_handle handle, int32* tag, int64*
                      value);

int32 pams_encode_uint64 ( pams_handle handle, int32* tag,
                      uint64* value);

int32 pams_encode_float ( pams_handle handle, int32* tag, float*
                      value);

int32 pams_encode_double ( pams_handle handle, int32* tag,
                      double* value);

int32 pams_encode_string ( pams_handle handle, int32* tag, char*
                      value, int32* length);
```

```
int32 pams_encode_array ( pams_handle handle, int32* tag, void*
                          value, int32* numElts);

int32 pams_encode_qid ( pams_handle handle, int32* tag,
                          q_address* value);
```

Arguments

The following table describes the arguments for the pams_encode functions. Some of these arguments apply to certain functions only.

Table 9-8

Argument	Data Type	Mechanism	Prototype	Access
`handle`	pams_handle	reference	`pams_handle` *	passed
`tag`	int32	reference	int32 *	passed
`value`	varying	reference	int32 *	passed
`length`	int32	reference	int32 *	passed
`numElts`	int32	reference	int32 *	passed

Argument Definitions

handle

Specifies the message handle.

tag

Specifies the tag of the field to encode.

value

Specifies the value of the field to encode.

length

Specifies the length of the string to encode, or -1 if the string is null terminated.

numElts

Specifies the number of elements in the array.

Return Values

Table 9-9

Return Code	Platform	Description
`PAMS__BADHANDLE`	All	Invalid message handle or handle to an untyped message.
`PAMS__BADTAG`	All	Invalid tag.
`PAMS__RESRCFAIL`	All	Insufficient resources to expand the message.
`PAMS__SUCCESS`	All	Indicates successful completion.

Description

Several structures exist to encode fields for each data type supported by the SDM capability. The application developer uses the function that matches the data type of the field to encode. Since the tag embeds information about the value data type, the `PAMS__BADTAG` code is returned if the function used does not match the tag data type. For example, if the pams_encode_int32 function is used to encode a character string. The `PAMS__BADTAG` code is also returned if the tag construction does not follow the rules or if the tag is reserved.

You can insert a null tag (`PSDM_NULL_TAG`) in a SDM message to control application behavior. For example, you can insert a null tag to stop an enumeration. To insert a null tag, use any of the numeric `pams_encode` functions and specify a NULL value pointer. The following code fragment shows how to insert a null tag into a signed 32-bit integer element:

```
null_tag = PSDM_NULL_TAG;
status = pams_encode_int32(mh, &null_tag, NULL);
```

See Also

- `pams_decode`

pams_extract_buffer

Returns a message in the specified buffer. The message size is also returned.

Syntax

```
int32 pams_extract_buffer ( handle, msgBuffer, bufferLength,
msgLength);
```

Table 9-10

Argument	Data Type	Mechanism	Prototype	Access
`handle`	`pams_handle`	reference	`pams_handle` *	passed
`msgBuffer`	char	reference	char *	returned
`bufferLength`	uint32	reference	uint32 *	passed
`msgLength`	uint32	reference	uint32 *	returned

Argument Definitions

handle

Specifies the message handle of the message to extract.

msgBuffer

Contains the pointer to the buffer from which to extract the message.

bufferLength

Specifies the size in bytes of `msgBuffer`.

msgLength

Contains the pointer where to place the message’s length. You can specify a NULL pointer in languages that allow these kinds of pointers.

Return Values

Table 9-11

Return Code	Platform	Description
`PAMS__AREATOSMALL`	All	Message is larger than the user’s buffer.
`PAMS__BADARGLIST`	All	Invalid number of arguments.
`PAMS__BADPARAM`	All	Invalid `bufferLength` argument.

Table 9-11

Return Code	Platform	Description
`PAMS__BADHANDLE`	All	Invalid message handle or handle to an SDM message already processed with the API.
`PAMS__FATAL`	All	SDM message is corrupted.
`PAMS__SUCCESS`	All	Indicates successful completion.

Description

This function copies the received message associated with the specified handle into the user provided buffer. If requested (by passing a non-NULL `msgLength`), the number of bytes of the message is returned as well.

If the message handle points to an SDM message for which encoding or decoding has already been performed, this function returns `PAMS__BADHANDLE`.

If your application does not know the maximum size message that can arrive, it can call the `pams_msg_length` function prior to calling `pams_extract_buffer` to determine how big a buffer is needed.

See Also

- `pams_create_handle`
- `pams_delete_handle`
- `pams_insert_buffer`
- `pams_msg_length`

pams_insert_buffer

Note: This function is obsolete. Handles and the SDM format used in MessageQ Version 4.0 have been replaced by the FML32 format for self describing messaging. Use `pams_put_msg` with the `PSYM_MSG_FML` symbol and `pams_get_msg(w)` with the `PSYM_MSG_BUFFER_PTR` symbol to send and receive FML messages.

Inserts the contents of the specified buffer into the message identified by the message handle.

Syntax

```
int32 pams_insert_buffer ( handle, msgBuffer, length );
```

Arguments

Table 9-12

Argument	Data Type	Mechanism	Prototype	Access
`handle`	`pams_handle`	reference	`pams_handle` *	passed
`msgBuffer`	char	reference	char *	passed
`length`	uint32	reference	uint32 *	passed

Argument Definitions

handle

Specifies the message handle.

msgBuffer

Specifies the pointer to a user area to use as message content.

length

Specifies the size in bytes of the `msgBuffer` buffer or zero.

Return Values

Table 9-13

Return Code	Platform	Description
`PAMS__BADARGLIST`	All	Invalid number of arguments.

Table 9-13

Return Code	Platform	Description
`PAMS__BADHANDLE`	All	Invalid message handle or handle to an SDM message and `*length > 0` or not handle to an SDM message and `*length == 0`.
`PAMS__BADPARAM`	All	Invalid `msgBuffer` or `length` argument.
`PAMS__FATAL`	All	SDM message is corrupted.
`PAMS__RESRCFAIL`	All	Insufficient resources to complete the operation.
`PAMS__SUCCESS`	All	Indicates successful completion.

Description

This function copies the user-provided buffer into the received message that is associated with the specified handle. If a buffer was already inserted in the message, it is overwritten by a subsequent call to the `pams_insert_buffer` function. If the `length` argument points to a zero-valued integer, the buffer to insert is an SDM message.

See Also

- `pams_create_handle`
- `pams_delete_handle`
- `pams_extract_buffer`
- `pams_msg_length`

pams_msg_length

Note: This function is obsolete. Handles and the SDM format used in MessageQ Version 4.0 have been replaced by the FML32 format for self describing messaging. Use `pams_put_msg` with the `PSYM_MSG_FML` symbol and `pams_get_msg(w)` with the `PSYM_MSG_BUFFER_PTR` symbol to send and receive FML messages.

Returns the number of bytes in the message. The message is identified by a message handle created with the `pams_create_handle` function.

Syntax

```
int32 pams_msg_length ( handle, msgLength );
```

Arguments

Table 9-14

Argument	Data Type	Mechanism	Prototype	Access
`handle`	`pams_handle`	reference	`pams_handle` *	passed
`msgLength`	uint32	reference	uint32 *	returned

Argument Definitions

handle

Specifies the message handle of the message.

msgLength

Contains the message handle of the message.

Return Values

Table 9-15

Return Code	Platform	Description
PAMS__SUCCESS	All	Indicates successful completion.
PAMS__BADARGLIST	All	Invalid number of arguments.
PAMS__BADHANDLE	All	Invalid message handle.

See Also

- pams_create_handle
- pams_delete_handle

- pams_extract_buffer
- pams_insert_buffer

pams_next_msg_field

Returns the tag and length of the first unseen field in the message.

Syntax

```
int32 pams_next_msg_field (pams_handle handle, tag, valueLength)
```

Arguments

Table 9-16

Argument	Data Type	Mechanism	Prototype	Access
handle	pams_handle	reference	pams_handle *	passed
tag	int32	reference	int32 *	returned
valueLength	int32	reference	int32 *	returned

Argument Definitions

handle

Specifies the message handle of the message to scan.

tag

Returns the tag of first unseen field in the message.

valueLength

Returns the length of the returned value or a NULL pointer, if the user is not interested in this information. The length returned is the size in bytes necessary to receive the value. For strings, it is the string length (null terminator not included). For arrays, it is the number of fields multiplied by the size of each field.

Return Values

Table 9-17

Return Code	Platform	Description
PAMS__BADHANDLE	All	Invalid message handle.
PAMS__NOMORETAG	All	No more tags in the message (all fields were decoded).
PAMS__SUCCESS	All	Indicates successful completion.

Description

This function does not mark the matched field as seen. Therefore, successive calls to this function without calling the appropriate pams_decode functions returns the same tag.

When all fields have been decoded, the function returns PAMS__NOMORETAG.

See Also

- pams_decode
- pams_set_msg_position

pams_remove_encoding

Note: This function is obsolete. Handles and the SDM format used in MessageQ Version 4.0 have been replaced by the FML32 format for self describing messaging. Use `pams_put_msg` with the PSYM_MSG_FML symbol and `pams_get_msg(w)` with the PSYM_MSG_BUFFER_PTR symbol to send and receive FML messages.

Removes a previously encoded field from the message buffer.

Syntax

```
int32 pams_remove_encoding (pams_handle handle, int32* tag, int32*
flags);
```

Argument

Table 9-18

Argument	Data Type	Mechanism	Prototype	Access
handle	pams_handle	reference	pams_handle *	passed
tag	int32	reference	int32 *	passed
flags	int32	reference	int32 *	passed

Argument Definitions

handle

Specifies the message handle.

tag

Specifies the tag of the field to remove. If the tag is PSDM_NULL_TAG, the first or last - depending on flags - encoded null tag is removed.

flags

Specifies the flags to control the function behavior. The flags argument can take the following values:

- PSDM_FIRST to remove the first encoded field matching tag.
- PSDM_LAST to remove the last encoded field matching tag.
- PSDM_ANY to ignore the tag argument; PSDM_ANY must be used in combination with PSDM_FIRST or PSDM_LAST, to force the very first or very last field in the message to be removed, whatever its tag is.

Return Values

Table 9-19

Return Code	Platform	Description
PAMS__BADPARAM	All	Invalid flag.
PAMS__BADHANDLE	All	Invalid message handle or handle to a large message.
PAMS__BADTAG	All	Invalid tag.
PAMS__SUCCESS	All	Indicates successful completion.
PAMS__TAGNOTFOUND	All	Tag not found in the message.

See Also

- pams_decode
- pams_encode

pams_set_msg_position

Note: This function is obsolete. Handles and the SDM format used in MessageQ Version 4.0 have been replaced by the FML32 format for self describing messaging. Use `pams_put_msg` with the PSYM_MSG_FML symbol and `pams_get_msg(w)` with the PSYM_MSG_BUFFER_PTR symbol to send and receive FML messages.

Resets the message to the position of a specific tag.

Syntax

```
int32 pams_set_msg_position (pams_handle handle, int32* tag, int32*
                        flags);
```

Arguments

Table 9-20

Argument	Data Type	Mechanism	Prototype	Access
handle	pams_handle	reference	pams_handle *	passed
tag	int32	reference	int32 *	passed
flags	int32	reference	int32 *	passed

Argument Definitions

handle

Specifies the message handle.

tag

Specifies the tag to reset encoding or decoding to.
Specifies the flags argument, which is a mask allowing you to specify the behavior. It can OR the following modifiers:

- PSDM_PREVIOUS searches for the previous occurrence of the specified tag (default for encoding).
- PSDM_NEXT searches for the next occurrence of the specified tag (default for decoding).
- PSDM_FIRST searches for the first occurrence of the specified tag in the message.
- PSDM_LAST searches for the last occurrence of the specified tag in the message.

- PSDM_AT sets the position at the element specified (default).
- PSDM_BEFORE sets the position at the element preceding the specified element.
- PSDM_AFTER sets the position at the element following the specified element.
- PSDM_ANY tells to ignore the specified tag argument. The following combinations are possible :
- (PSDM_ANY | PSDM_PREVIOUS) sets position at the previous tag.
- (PSDM_ANY | PSDM_NEXT) sets position at the next tag; when used in conjunction with pams_next_msg_field, it allows skipping the undesired elements.
- (PSDM_ANY | PSDM_FIRST) sets position at the beginning of the message.
- (PSDM_ANY | PSDM_LAST) sets position at the end of the message; this allows appending elements to a received message.
  The modifiers PSDM_PREVIOUS, PSDM_NEXT, PSDM_FIRST, and PSDM_LAST are mutually exclusive.
  The modifiers PSDM_AT, PSDM_BEFORE, and PSDM_AFTER are also mutually exclusive.

Return Values

Table 9-21

Return Code	Platform	Description
PAMS__BADHANDLE	All	Invalid message handle or handle to a large message.
PAMS__BADPARAM	All	The specified flags argument is invalid.
PAMS__SUCCESS	All	Indicates successful completion.

Description

This function resets the starting point of the encoding or decoding to the field in the message with the specified tag:

- When used to perform further encoding, already encoded fields after the specified element are lost.

- When used to perform further decoding, all seen fields after the specified element are marked unseen and the rest are marked seen.

See Also

- pams_encode
- pams_decode
- pams_next_msg_field

Index