DECmessageQ

Introduction to Message Queuing

Order Number: AA–PMDVD–TK

April 1994

This guide describes how DECmessageQ message queuing software integrates distributed applications in a multiplatform environment.

Revision/Update Information: This is a revised document.

Software Version:

DECmessageQ for 88open, Version 3.0
DECmessageQ for AIX, Version 3.0
DECmessageQ for HP-UX, Version 3.0
DECmessageQ for Macintosh, Version 2.0
DECmessageQ for MS–DOS, Version 2.0
DECmessageQ for OpenVMS AXP, Version 2.0
DECmessageQ for OpenVMS VAX, Version 2.1
DECmessageQ for OS/2, Version 2.0
DECmessageQ for SunOS, Version 3.0
DECmessageQ for Solaris, Version 3.0
DECmessageQ for ULTRIX RISC, Version 3.0
DECmessageQ for ULTRIX VAX, Version 3.0
DECmessageQ for Windows NT, Version 3.0

Digital Equipment Corporation
Maynard, Massachusetts

April 1994

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

© Digital Equipment Corporation 1991, 1994.
All rights reserved.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to help prepare future documentation.

The following are trademarks of Digital Equipment Corporation:

AXP, CDD, DEC, DECmessageQ, DECnet, DECstation, OpenVMS, PATHWORKS, RX, TK, ULTRIX, VAX, VAX C, VAXELN, VAXstation, VMScluster, and the DIGITAL logo.

88open is a trademark of the 88open Consortium, Ltd. AppleShare, AppleTalk, Macintosh, and MacTraps are registered trademarks of Apple Computer, Inc. Hewlett-Packard and HP are registered trademarks and 9000/300, 9000/400, 9000/700, 9000/800, and HP-UX are trademarks of Hewlett-Packard Company. AIX, IBM, and OS/2 are registered trademarks of International Business Machines Corporation. Motif and OSF/1 are registered trademarks of Open Software Foundation, Inc. Motorola is a registered trademark of Motorola, Inc. MS–DOS, Windows, and Windows NT are registered trademarks and Excel is a trademark of Microsoft Corporation. NFS, Solaris, Sun, and SunOS are registered trademarks of Sun Microsystems, Inc. PostScript is a registered trademark of Adobe Systems, Incorporated. SPARC is a trademark of Sparc International, Inc. THINK C is a trademark of Symantec Corporation. UNIX is a registered trademark of UNIX System Laboratories, Inc. a wholly-owned subsidiary of Novell, Inc. Xerox is a registered trademark of Xerox Corporation.

All other trademarks and registered trademarks are the property of their respective holders.

This document is available on CD-ROM.

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

Preface v

1 Integrating Distributed Applications

Evolution of Distributed Computing 1–1
Trend Toward Client/Server Computing 1–2
Integration Through Message Queuing 1–4
Support for Multivendor Environments 1–5

2 Message Queuing Concepts

Message Queuing Basics 2–1
DECmessageQ Architecture 2–3
Message Broadcasting 2–6
Message Selection 2–8
Message Recovery 2–9

3 Programming Services

Application Programming Interface (API) 3–1
DECmessageQ API Services 3–2
Testing and Debugging 3–5

4 DECmessageQ System Management

System Configuration 4–1
Monitoring System Performance 4–2
Troubleshooting System Problems 4–3

Glossary

Index

Figures

1–1	Application Distribution in Client/Server Computing	1–3
1–2	DECmessageQ Multivendor Support	1–6
2–1	Traditional Integration Versus Message Queuing	2–4
2–2	DECmessageQ Message Broadcasting	2–6
3–1	Common API for All Platforms	3–1
4–1	Monitor Utility—Motif UI	4–2

Tables

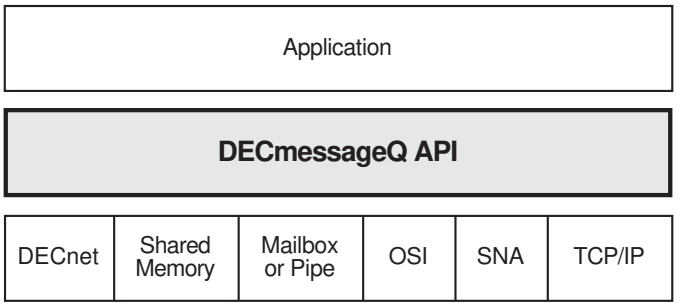
1–1	Need for Integrating Distributed Applications	1–2
1–2	Benefits of Using DECmessageQ	1–5
2–1	Support for Message Queue Types	2–6
2–2	Phases of SBS Broadcasting	2–7
2–3	Support for Message Broadcasting	2–8
2–4	Message Recovery Files	2–10
2–5	Support for Message Recovery	2–11
3–1	Support for API Services	3–4
3–2	Support for Message Scripts	3–6

Preface

What is DECmessageQ

DECmessageQ software is a programming tool that enables applications to send and receive messages among networked systems through queued messaging. You can use DECmessageQ software to integrate applications running on Windows and Windows NT PCs, UNIX and OpenVMS workstations and servers, Macintosh systems, and IBM mainframes.

DECmessageQ uses a single application programming interface (API) for all platforms which allows applications to port easily between environments. Applications using DECmessageQ as a common communication mechanism are insulated from changes in operating system and network software.



MK–2468–GE

DECmessageQ provides a message recovery system to guarantee message delivery in the event of system, process, and network failures. In addition, it offers message broadcasting, message simulation, capture, and replay, and utilities for system management and application debugging.

Why Use DECmessageQ

DECmessageQ software enhances:

- **Productivity** by providing a standard integration approach that speeds development and insulates applications from changes in network and operating system software.
- **Portability** by offering an application programming interface for building applications that run in all industry-leading environments.
- **Simplicity** by providing a common communication mechanism for distributed applications to send and receive messages.
- **Reliability** by offering guaranteed message delivery in the event of system, process, and network failures.
- **Interoperability** by integrating UNIX, Windows NT, OpenVMS, Macintosh, MS–DOS, and OS/2 applications.

What Is in Your Kit

In addition to your DECmessageQ license certificate, you receive a distribution kit containing:

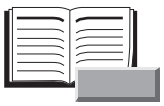
Media	The distribution media appropriate for your system.
SPD/SSA	*Software Product Description* and *System Support Addendum* describing the supported environments.
Cover letter	Instructions for printing late-breaking news about this release.
Documentation	*Introduction to Message Queuing* *Installation and Configuration Guide* *Programmer's Guide* *Online help*

Where to Call for Help

DECmessageQ software is covered by a one-year software warranty. If you experience a problem with DECmessageQ, you can call your country support center for assistance. In the U.S., call Digital's Customer Support Center at 1-800-DIGITAL (1-800-344-4825) 24 hours a day, 7 days a week. Please be sure to have your DEC number ready when you call.

In addition to the one year warranty, Digital offers a variety of software support options for DECmessageQ. To obtain more information about purchasing a support contract, contact your local Digital Sales Representative.

Conventions Used in This Guide



The following conventions are used in this guide:

boldface type	Boldface type is used to distinguish routine call arguments and command parameters when they are discussed in text. New terms, also shown in boldface type, are defined in the glossary in *DECmessageQ Introduction to Messaging*.
`system output`	This typeface indicates system output or the exact name of a command, option, partition, pathname, directory, or file.
italic type	Italic type emphasizes important information, indicates variables, and indicates complete titles of manuals.
UPPERCASE	Words in uppercase indicate a command, the name of a file, the name of a file protection code, or an abbreviation for a system privilege.
lowercase	In format descriptions, words in lowercase indicate parameters or arguments to be specified by the user.
n	A lowercase italic *n* indicates the generic use of a number. For example, 19*nn* indicates a 4-digit number in which the last 2 digits are unknown.
[ ]	In format descriptions, brackets indicate optional elements. Brackets are not optional, however, in the syntax of an OpenVMS directory name or file specification.
. . .	The vertical ellipsis indicates the omission of information from an example or command format.
$	This symbol indicates the default DCL prompt.
#	This symbol indicates the default superuser prompt.
Return	Press the key labeled Return on the keyboard.
Enter	Type the information that follows and press the Return key.

1
Integrating Distributed Applications

Distributed computing, an important computing trend that evolved during the last several decades, sought to move application processing close to its user base. **Distributed applications** allow more control at the user and department level, increase the accuracy and timeliness of data collection, and enable greater use of data by end users.

However, the evolution of distributed computing has created new challenges for information technology (IT) professionals. Departmental and end-user solutions can often be isolated and unable to share data with other corporate systems. The autonomy of departmental and end-user solutions led to the purchase of hardware and software from different vendors, resulting in a heterogeneous computing environment. And, while distributed access to corporate data is needed, IT departments must still develop systems that protect the company's valuable information assets.

This introduction to DECmessageQ software describes new trends in distributed computing targeted to meet today's challenges in information systems design. It describes the need for integrating distributed applications and how **message queuing** can play a key role in integrating multivendor environments.

Evolution of Distributed Computing

With the advent of smaller, more powerful computers, companies have increasingly decentralized responsibility for implementing application solutions. During the past two decades, many departments and end users implemented application solutions that were very efficient for part of the business, though they were not designed to meet the larger needs of the company.

Integrating Distributed Applications

Today many businesses analyze their needs on a global, enterprisewide basis. They now realize that the ability to share data automatically between users, departments, divisions, and locations increases the reliability and timeliness of information they use to shape their business decisions.

Table 1–1 lists some of the major reasons why companies are integrating their distributed applications.

Table 1–1 Need for Integrating Distributed Applications

√	Companies need to...
_ _	Eliminate manual data entry by sharing data automatically between departmental systems
_ _	Exchange information between applications at remote sites
_ _	Employ diverse computer hardware for different applications
_ _	Consolidate business operations to reduce redundancies and control cost
_ _	Coordinate application processing to promote efficiency
_ _	Consolidate reporting from different information sources within the company
_ _	Reduce software development overhead
_ _	Provide distributed access to data stored on corporate mainframes
_ _	Connect hundreds of PCs across the company to share data
_ _	Ensure reliable exchange of information in a diverse environment

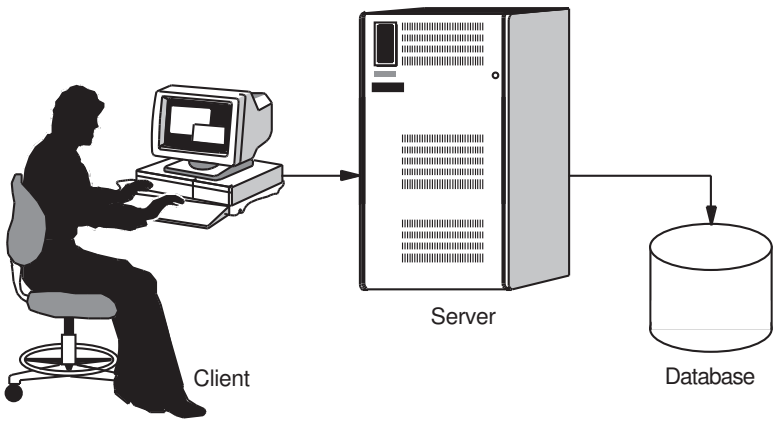
Trend Toward Client/Server Computing

The advent of powerful desktop computers has created greater access to corporate information by an increasing number of sophisticated users. At the same time, IT departments need to maximize the use of computer resources while protecting valuable corporate information assets. The **client/server model** has emerged as a new approach to distributed application design that provides shared and efficient access to corporate information resources with appropriate safeguards.

Typically, applications consist of the following components: a **user interface (UI)**, a computational function, and a database of information. Traditional application design can place all application components on the same computer, restricting responsiveness and isolating data. However, designing a system

that allows these same application components to run on different computer systems can maximize the efficiency of computer systems while distributing user access to a corporate information base. Figure 1–1 shows how the components of a client/server application can be distributed among multiple systems.

Figure 1–1 Application Distribution in Client/Server Computing



The client/server model distributes application components for maximum efficiency. For example, the UI component is generally placed on client systems—desktop devices that provide better response time for users and offload larger systems from the high volume of user activity. The client/server model places major application functions such as computation and database storage on server systems—larger computers equipped with greater computing and storage resources.

Decomposing an application into its component parts and distributing the parts across disparate computer systems is much more complex than implementing an application on a single system. Therefore, a standard software integration mechanism that works across multiple platforms can represent a significant time and cost savings to programming teams.

Integration Through Message Queuing

The challenge to developers when integrating distributed applications is to manage the interdependencies between application programs. Interdependent applications can run on the same system or different computers in a network. To manage these interdependencies, applications need a common mechanism for exchanging information. **Message queuing** enables applications to:

- Request functions or services
- Exchange information
- Synchronize communication

Message queuing software forms a layer in between the application and the underlying operating system and network software which is referred to as middleware. Without middleware to accomplish a message exchange, application developers have to write the software for sending and receiving information by learning how to use the features of both network and operating system software to transport the message. And, without a standard approach to message queuing, each application must be programmed to communicate with each of the other applications in the multiplatform environment.

For example, to exchange local messages on OpenVMS systems, you can use OpenVMS mailboxes, though message exchange on an OSF/1 AXP system requires knowledge of queues or pipes. Communicating between remote OpenVMS and OSF/1 AXP systems requires knowledge of DECnet task-to-task communication or TCP/IP socket programming.

DECmessageQ software enables the exchange of information through message queuing. To communicate using DECmessageQ, applications are associated with a **message queue** attached to a **message queuing bus**. Messages are sent to message queues to be read by applications. The DECmessageQ message queuing bus provides a standard mechanism for many applications to share information by sending and receiving messages.

Table 1–2 lists the benefits of using the DECmessageQ message queuing bus for multivendor application integration.

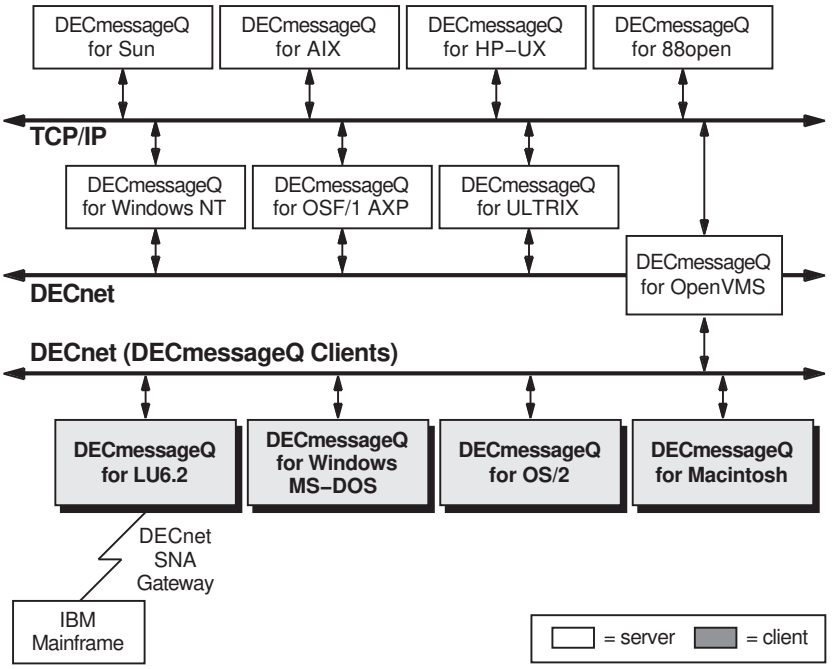
Table 1–2 Benefits of Using DECmessageQ

Productivity	DECmessageQ provides a standard integration approach that speeds development and insulates applications from changes in network and operating system software.
Portability	DECmessageQ offers an application programming interface (API) that is common to all environments. Applications are written once, then ported to other systems.
Simplicity	The DECmessageQ message queuing bus provides a common communication mechanism for distributed applications to send and receive messages.
Reliability	DECmessageQ offers guaranteed message delivery in the event of system, process, and network failures.
Interoperability	DECmessageQ integrates applications running on all Digital Alpha AXP platforms: OSF/1, Windows NT, and OpenVMS. In addition, DECmessageQ integrates applications running on many popular industry platforms include Windows NT Intel systems, IBM AIX, HP-UX, SunOS, Solaris, 88open, RISC ULTRIX, VAX ULTRIX, OpenVMS, Windows, MS–DOS, OS/2, and Macintosh environments. It also provides integration of IBM host applications through the LU6.2 protocol.

Support for Multivendor Environments

DECmessageQ software uses a client/server architecture to implement distributed communication. Each configuration requires a message routing system, or message server, to distribute messages between remote nodes. Client systems can exchange messages locally, or remotely with other applications on the message queuing bus through a message server. Figure 1–2 shows the multivendor environment currently supported by DECmessageQ software.

Figure 1–2 DECmessageQ Multivendor Support



DECmessageQ offers message servers for all Digital Alpha AXP platforms: OSF/1, Windows NT, and OpenVMS. In addition, message servers are available for Digital's OpenVMS VAX, ULTRIX VAX, and ULTRIX RISC platforms. All DECmessageQ messages servers that run on Digital platforms offer message routing for TCP/IP and DECnet networks.

In addition to supporting Digital platforms, DECmessageQ offers message servers for popular UNIX platforms, including IBM AIX, Hewlett-Packard HP-UX, Sun Microsystems SunOS and Solaris platforms, and 88open compliant systems. These servers route messages using TCP/IP networking software.

DECmessageQ for OpenVMS complements client implementations for Microsoft MS–DOS, IBM OS/2, Apple Macintosh, and Digital ULTRIX environments. These clients use the DECmessageQ for OpenVMS message server to communicate using DECnet networking software. In addition, the DECmessageQ for OpenVMS message server acts as a message router between

Integrating Distributed Applications

DECmessageQ clients and the DECmessageQ message servers for UNIX and Windows NT systems.

DECmessageQ enables communication with applications residing on IBM host systems through the LU6.2 protocol. DECmessageQ software supporting the LU6.2 protocol requires IBM to Digital host communication via DECnet/SNA Gateway products.

2
Message Queuing Concepts

To help you understand more about how DECmessageQ software enhances programming productivity and protects corporate software investments, this chapter describes basic message queuing concepts, the DECmessageQ message queuing bus architecture, and DECmessageQ advanced message queuing capabilities.

Message Queuing Basics

Message queuing is the deliberate transmission of data between two or more cooperating processes. These cooperating processes can run on the same processor or on several processors in a single multiprocessing central processing unit (CPU). The process can also run on several CPUs running in a VMScluster configuration, across a local area network (LAN), or across a wide area network (WAN).

What Is a Message?

A **message** is an application-defined data structure. The application developer defines the content of the message. A message has the following components:

- Data that is defined by the application
- Message attributes that can be used by the application
- Message context defined by the message queuing software that is transparent to the user

A message queuing system provides peer-to-peer communication through a standard message-passing mechanism. Communicating programs can operate independently while using the message queuing system to exchange information. One program initiates communication with a remote program and exchanges messages with it, enabling two-way communication. Data and control information can flow in either direction. And, the communication

can be **synchronous** or **asynchronous**. That is, the sender program can wait for a reply from the receiver program or continue immediately.

Communication through a message queuing system requires both programs to agree upon the type of data in the message and the interpretation of the data. The message queuing system ignores the body of a message; its job is simply to transports the message data.

What Is Message Queuing?

Message queuing is a technique for sending messages from one program to another by directing messages to a memory- or disk-based queue as an intermediate storage point. The queue stores the messages until they can be processed by the receiver program. By queuing messages, programs can execute independently—they do not wait for an application to process a message before continuing.

Message queuing works in the following way:

1. Program A makes a call to the message queuing system. The call tells the message queuing system that a message is ready to be sent to Program B.
2. The message queuing system sends the message to the system where Program B resides and places it in Program B's primary queue.
3. In time, Program B reads and processes the message.

If the system cannot deliver the message because of a communications failure, process abort, or system crash, message recovery capabilities enable the message to be re-sent without further application intervention when communication is reestablished.

Using message queuing, any program can send messages addressed to any other program that is attached to the message queuing bus. The sender program sends out the messages and can continue processing because a response is not usually required. For this reason, message queuing adapts well to asynchronous interprocess communication needs.

Message queuing does not require applications to know the structure or state of another application in order to enable communication. For example, they do not have to share the same concepts of recovery and consistency. As a result, queued

communication offers a practical way to integrate applications running in heterogeneous environments.

DECmessageQ Architecture

DECmessageQ is an implementation of a message queuing system. To communicate using DECmessageQ, applications use the DECmessageQ message queuing bus. Applications are assigned a message queue to receive messages. Message queues are assigned a group in order to efficiently share system resources. **Cross-group** connections enable applications to share information across different systems connected to the same message queuing bus.

DECmessageQ Message Queuing Bus

DECmessageQ software uses a message queuing bus as a data highway to transfer messages between processes. By communicating through a message queuing bus, sender and receiver programs do *not* have to:

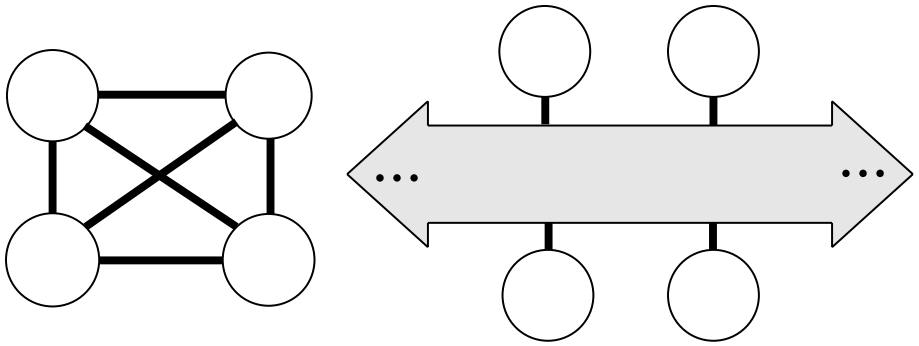
- Know each other's physical location (node name)
- Know how to establish communications between each other
- Be executing at the same time

The DECmessageQ message queuing bus provides a standard set of callable services that allow a programmer to transfer messages between processes on the same computer system or different computer systems. Each process using DECmessageQ software is required to **attach** to the message queuing bus. Once attached, the process can send messages to other processes attached to the same message queuing bus.

Communication between processes in the traditional model is extremely complex compared to the simple communication method offered by the message queuing bus. Figure 2–1 contrasts point-to-point communications and the message queuing bus approach.

Message Queuing Concepts

Figure 2–1 Traditional Integration Versus Message Queuing



DECmessageQ allows the configuration of more than one message queuing bus. Applications attached to different message queuing buses cannot communicate with each other. This feature allows users to set up a bus for communication of test programs and another bus for production applications. Critical application processing is then separated from the testing environment. Test programs are easily moved into production by simply changing their **bus ID**—a configuration step in using DECmessage that is external to the application.

Message Queuing Groups

A **message queuing group** defines a collection of queues sharing DECmessageQ resources. These queues are available for one or more application programs. Each DECmessageQ group is identified by a unique number, the DECmessageQ **group ID**. This group ID is part of the **queue address**.

A message queuing group encompasses related applications or components of the same application. Applications can utilize a single or multiple message queuing groups. The application designer decides how to distribute the application messaging among the groups.

Message Queues

A message queue is a resource configured by the application developer and managed by DECmessageQ. All messages sent with DECmessageQ are written to a queue. Each queue defined within DECmessageQ is identified by a number. This number is part of the queue address.

Application programs receiving DECmessageQ messages must be associated with at least one queue. The application program becomes associated with a queue when it attaches to the DECmessageQ message queuing bus.

Once the application is attached, it becomes the owner of the queue. Only the owner can read messages from the queue. The only exception to this rule is for multireader queues which can have many owners.

An application program can read messages from multiple queues. When a message is read from a queue, it is moved from the queue into a data **buffer** defined by the program. Once read, the message no longer exists on the queue. Unless message selection criteria are specified, messages are read from queues in first-in first-out (FIFO) order.

DECmessageQ software uses three **queue types** as follows:

- **Primary queue**—assigned to each process that attaches to the DECmessageQ message queuing bus. This queue is required in order to receive messages from processes using DECmessageQ.
- **Secondary queue**—used by processes in addition to the primary queue. The order in which queues are scanned for messages is defined by the DECmessageQ selections rules.
- **Multireader queue (MRQ)**—used to store messages that can be read by many simultaneous readers.

Table 2–1 shows the multiplatform support for various DECmessageQ queue types.

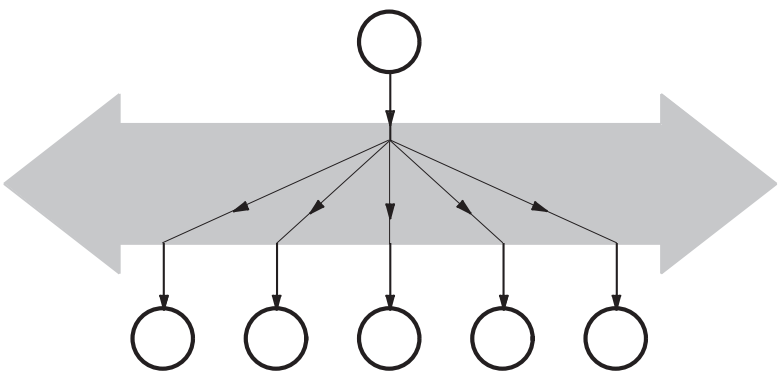
Table 2–1 Support for Message Queue Types

	Servers			Clients
Feature	UNIX	Windows NT	OpenVMS	MS–DOS, OS/2, Macintosh
Primary queue	X	X	X	X
Secondary queue	X	X	X	.
Multireader queue	X	X	X	.
Non-FIFO queue	X	X	X	.
Response queue	X	X	X	.
Max queues per group	999	999	999	1

Message Broadcasting

Message **broadcasting** is a style of communicating that uses one message sender program and multiple message receiver programs. DECmessageQ **selective broadcast services (SBS)** manages the broadcasting of data between processes and groups of processes. Figure 2–2 illustrates message broadcasting on the DECmessageQ message queuing bus.

Figure 2–2 DECmessageQ Message Broadcasting



MK–0128–GE

DECmessageQ broadcast services provide applications with:

- One-to-many message queuing

- Lists of application processes that are interested in messages that broadcast
- User-definable rules, known as selection rules, which can be used to selectively extract messages from a broadcast stream

A **broadcast stream** is a message pipeline that has a single entry point and multiple exit points. A message inserted into a broadcast stream flows through the pipeline until it reaches an exit point, then it is written to the message queues of the receiver programs. Because multiple exit points can exist on a broadcast stream, messages sent to the broadcast stream are simultaneously distributed to all registered queues.

The entry point to a broadcast stream is called a **Multipoint Outbound Target (MOT)**. A MOT is a queue number that is defined by a predefined range of numbers. An application program that wants to receive messages from a broadcast stream must send a registration message to the SBS Server to associate its queue number with the broadcast stream's exit point. The registration message contains the MOT and the queue number. This allows message broadcasting system to associate the MOT with the queue number of the receiver program.

Table 2–2 shows the phases of message broadcasting.

Table 2–2 Phases of SBS Broadcasting

1. Registration	An application program wanting to receive messages from a broadcast stream subscribes to the broadcast stream by sending a registration message to the SBS Server.
2. Submission	An application program wanting to broadcast a message places it on the broadcast stream for distribution. Selective broadcast services allows a program to define multiple broadcast streams.
3. Distribution	Selective broadcast services delivers the message to multiple receiver programs.
4. Deregistration	A queue is disconnected from the broadcast stream.

Message broadcasting simplifies the coding requirements because sender programs do not need to be aware of the number, state, and location of the distribution targets. Message broadcasting

increases efficiency by directing messages to many targets with a single call. However, message broadcasting does not support recoverable delivery modes.

SBS Servers run only on OpenVMS systems. Therefore, applications running on UNIX, Windows NT, Windows, MS–DOS, Macintosh, and OS/2 systems must use a cross-group connection to send and receive broadcast messages using an OpenVMS system as the distributor. To receive broadcast messages, applications must know the queue number of the SBS Server distributing the messages for a broadcast stream. To send broadcast messages, applications must know the MOT address on which they want to distribute their messages.

Table 2–3 shows the selective broadcast services supported by DECmessageQ in a multiplatform environment.

Table 2–3 Support for Message Broadcasting

	Servers			Clients
Feature	**UNIX**	**Windows NT**	**OpenVMS**	**MS–DOS, OS/2, Macintosh**
Server/Distributor	.	.	X	.
Sender/Receiver	X	X	X	X
Ethernet transport	.	.	X	.

Message Selection

DECmessageQ software allows application developers to use message selection filters when reading queued messages. These filters enable applications to selectively read messages by message type, class, source, or by a combination of selection criteria.

Message selection by message type and class allows application developers to assign classifications to messages and then selectively read messages of a particular type or class. In addition, applications can be programmed to read messages from a selected queue type or message source. The application specifies the selection criteria that must be matched for DECmessageQ to read the message. Full message selection is available for all DECmessageQ message server implementations.

In addition to message selection, DECmessageQ uses prioritized queuing to allow applications to read high priority messages first. Each queued message can be labeled with a high priority (1) or a low priority (0). Applications can then request that DECmessageQ deliver priority 1 messages before priority 0 messages. Using priority queuing, applications always read time-critical messages before less critical ones because they are placed at the top of the queue.

Message Recovery

When an application sends a message across a communications network, the final receipt of the message can be interrupted by various failure conditions. DECmessageQ supports the following basic options for recoverable and nonrecoverable message delivery:

- **Datagram**—A nonrecoverable attempt is made to deliver a message. If the message cannot be delivered to a target, then an error is logged.
- **Wait for enqueue**—The sending process will block until the message is written to the *target queue*. A return status indicates if the message is successfully written to the queue.
- **Return-to-sender**—A nonrecoverable attempt is made to deliver the message. If the message cannot be delivered, it is returned to the sending process marked with a special return status.
- **Store and forward (SAF)**—The sending process will send messages that are stored in a disk file locally before they are sent over any communications link.
- **Journaled guaranteed delivery**—The sending process will send messages that will be delivered to the target's disk queue.
- **Wait for dequeue**—The sending process will block until the message is read from the target queue by the receiver program.

By choosing a recoverable delivery option, DECmessageQ stores the message on a nonvolatile disk to be delivered as soon as it is possible. DECmessageQ implements these options through its **Message Recovery Services (MRS)**. With MRS, applications

can recover from message delivery failures caused by any of the following:

- Communications failure
- Application task abort
- System crash—sender, receiver, or both

To provide message recovery capabilities, messages are automatically backed up during communications or system failure and re-sent when communication is reestablished. For the sender program, MRS provides four files for storing, recovering, and resending messages as shown in Table 2–4.

Table 2–4 Message Recovery Files

File	Description
Store and forward (SAF)	Automatically resends messages after a loss of communications links across the network.
Destination queue file (DQF)	Automatically resends messages due to failures in the receiver program or system.
Dead letter journal (DLJ)	Allows the application or system manager to handle message redelivery.
Postconfirmation journal (PCJ)	Stores successfully delivered messages forming a log of message processing.

When the SAF and DQF recovery options are selected, the message recovery is automatic. If a message cannot be delivered due to a failure condition, it is stored and re-sent at a later time.

When an application receives and reads a recoverable message, it must use the pams_confirm_msg service to confirm message delivery. Confirming the delivery removes the message from the message recovery system. If the message is not confirmed, it will be redelivered if the application detaches and reattaches to the queue.

Applications can request that messages be written to the DLJ file if they cannot be stored by the message recovery system. Then the applications can re-send the messages from the DLJ file at a later time. The PCJ file, on the other hand, provides an audit trail of successfully delivered and confirmed messages.

Messages are read from the DLJ and PCJ files using DECmessageQ utilities or programming services. On DECmessageQ UNIX and Windows NT systems, the Journal Replay utility resends messages stored in the DLJ and PCJ files. The Journal Dump utility prints messages stored in the DLJ and PCJ files. On DECmessageQ OpenVMS systems, the pams_open_jrn, pams_read_jrn, and pams_close_jrn services are used to replay the messages in the journal files.

Table 2–5 shows the types of recoverable and nonrecoverable message delivery supported by DECmessageQ in a multiplatform environment.

Table 2–5 Support for Message Recovery

	Servers			Clients
Feature	**UNIX**	**Windows NT**	**OpenVMS**	**MS–DOS, OS/2, Macintosh**
Automatic recovery	X	X	X	.
pams_confirm_msg	X	X	X	.
Journal Replay	X	X	.	.
Journal Dump	X	X	.	.
pams_open_jrn	.	.	X	.
pams_read_jrn	.	.	X	.
pams_close_jrn	.	.	X	.

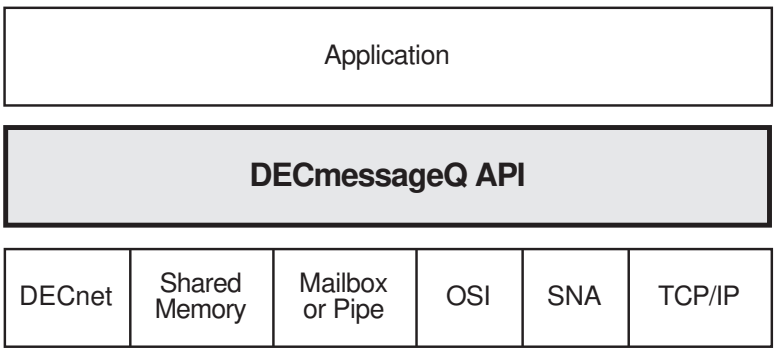
3
Programming Services

DECmessageQ offers code portability to many different environments through its standard set of programming services. This chapter describes the DECmessageQ **application programming interface (API)**, its programming services, and the DECmessageQ Script Facility for testing and debugging application programs.

Application Programming Interface (API)

The DECmessageQ API is a set of programming services that support the exchange of information in a multivendor environment. The DECmessageQ API separates application programs from the network and operating system mechanisms used to exchange messages as shown in Figure 3–1.

Figure 3–1 Common API for All Platforms



MK–2468a–GE

Programming Services

Using a single API for all environments, developers are freed from learning different operating system and network software. Instead they use one consistently implemented product for exchanging application messages.

Since DECmessageQ software handles the system level of message exchange, a large portion of programming time is eliminated from developing distributed software. Using a standard message queuing product lowers maintenance costs by replacing custom software with a vendor product. A common API also makes applications portable between environments.

DECmessageQ API Services

The DECmessageQ API offers services for sending and receiving messages that are common to all DECmessageQ environments. These services enable applications to attach to the message queuing bus, send and receive a message, and detach from the message queuing bus.

In addition, DECmessageQ has services for creating complex message selection criteria, confirming recoverable message delivery, creating timers, obtaining the number of pending messages for a queue, and sending messages asynchronously (OpenVMS systems only).

This section provides a brief description of each DECmessageQ programming service. For a detailed description of each service, refer to the *DECmessageQ Programmer's Guide*.

Attaching	The pams_attach_q service allows a process to attach to the DECmessageQ message queuing bus at a specified queue. The pams_attach_q call must be successfully executed before an application can send and receive messages.
Sending messages	The pams_put_msg service captures the contents of the user's buffer and sends it to a target queue or a broadcast stream. Special parameters define how and where to send a message and label it with selection criteria.
Receiving messages	DECmessageQ provides two alternative methods for receiving messages. You can have the process wait indefinitely, or you can set an optional timeout parameter.

	The pams_get_msg service reads the next message from the queue and moves the message to the user's local buffer. (The message that pams_get_msg reads from the queue must first match any established message selection criteria.) If the queue does not contain a message, pams_get_msg does not wait for another message, but returns a NOMOREMSG status.
	The pams_get_msgw service reads the next message from the queue and moves the message to the user's local buffer. This message is the same as pams_get_msg with the following exception: if the queue does not contain a message, pams_get_msgw waits for another message to enter the queue.
Detaching and Existing	The pams_detach_q service can detach a selected message queue or all of the application's message queues from the message queuing bus. The pams_exit service detaches the process and all of its message queues from the DECmessageQ message queuing bus.
Locating a queue	When dynamic naming is used, DECmessageQ software can translate symbolic names during program execution to reveal the queue address. This feature offers more flexibility than symbolic naming, which requires translation at compile time.
	The pams_locate_q service obtains a queue address for a specified queue name. The queue address must be known before the application can send a message to the target queue.
Using timers	DECmessageQ provides a timer capability to notify applications that a specified amount of time has elapsed or a time of day has arrived. This feature simplifies the implementation of periodic actions by eliminating the need to write application-specific timer code.
	The pams_set_timer service sets a timer to expire after a stated time interval or when a time of day arrives. When the timer goes off, a priority 1 message is sent to the source queue, allowing the timer expiration message to be received through the pams_get_msgw service.

Programming Services

	The pams_cancel_timer service cancels all DECmessageQ timers with a specified timer identification. All expired timers with the same ID that are waiting in the queue are purged and are not delivered.
Confirming recoverable message delivery	The pams_confirm_msg service informs the MRS Server that the receiver program has processed the recoverable message. If journaling is enabled, every confirmed message is written to the **journal file**.
Obtaining the number of pending messages for a queue	The putil_show_pending service requests the number of pending messages for a list of selected queues. The value returned by this service contains the total number of messages in each memory queue as well as the number of messages in the local and remote recovery journals targeted for delivery to the selected queue.
Receiving messages asynchronously	DECmessageQ provides mechanisms for writing interrupt-driven code.
	The pams_get_msga service is a special form of the pams_get_msg call that allows multiple asynchronous read operations with full selective reception. DECmessageQ interrupts the application when a message enters the queue, and executes the action routine specified in the call.
	The pams_cancel_get service cancels all pending pams_get_msga requests that match the selection filter.

Table 3–1 shows the multivendor support for DECmessageQ programming services.

Table 3–1 Support for API Services

	Servers			Clients
Feature	**UNIX**	**Windows NT**	**OpenVMS**	**MS–DOS, OS/2, Macintosh**
pams_attach_q	X	X	X	X
pams_cancel_get	.	.	X	.

(continued on next page)

Table 3–1 (Cont.) Support for API Services

	Servers			Clients
Feature	**UNIX**	**Windows NT**	**OpenVMS**	**MS–DOS, OS/2, Macintosh**
pams_cancel_select	X	X	X	.
pams_cancel_timer	X	X	X	.
pams_detach_q	X	X	.	X
pams_exit	X	X	X	X
pams_get_msg	X	X	X	X
pams_get_msga	.	.	X	.
pams_get_msgw	X	X	X	X
pams_locate_q	X	X	X	.
pams_put_msg	X	X	X	X
pams_set_select	X	X	X	.
pams_set_timer	X	X	X	.
putil_show_pending	X	X	X	.

Testing and Debugging

The DECmessageQ **script facility** helps developers test message exchange among application modules using DECmessageQ. Using the script facility, applications developers can:

- Simulate messages from an application not yet developed in order to test and debug an existing application
- Selectively capture application messages
- Replay messages to perform repetitive application testing

Message simulation offers a shortcut for sending messages to an application. Instead of writing a program to send a message, you can use a text editor to create a **script file**. The script file contains the message information and other instructions such as whether to log the message exchange.

Programming Services

The message information and other instructions are entered to the script file using the DECmessaageQ scripting language. When script processing is enabled, DECmessageQ processes the contents of the script file and delivers the message to the target queue where it can be read by the receiver program.

Message capture provides a mechanism for viewing the messages sent or received by an application. To capture messages, use the scripting language to create a script file that identifies the messages to be captured. Captured messages can be displayed on the screen or written to a log file. When script processing is enabled, DECmessageQ captures the messages and displays or logs them as specified in the script file.

Message replay uses the messages captured in a log file as input to an application in exactly the same way as messages entered to a script file. The script replay feature lets developers capture messages sent or received by an application and supply them as input to another program. By tracking the program's response to the captured messages, the developer can debug message exchange between programs that share information using DECmessageQ.

Note

The DECmessageQ Script Facility is available on UNIX and OpenVMS systems only.

Table 3–2 shows the multivendor support for DECmessageQ message capture, simulation, and replay capabilities.

Table 3–2 Support for Message Scripts

	Servers			Clients
Feature	UNIX	Windows NT	OpenVMS	MS–DOS, OS/2 Macintosh
Capture	X	X	X	X
Simulation	X	.	X	.

(continued on next page)

Table 3–2 (Cont.) Support for Message Scripts

	Servers			Clients
Feature	**UNIX**	**Windows NT**	**OpenVMS**	**MS–DOS, OS/2 Macintosh**
Replay	X	.	X	.
Script Verify	X	.	X	.

4

DECmessageQ System Management

This chapter describes DECmessageQ system configuration and the tools for monitoring DECmessageQ systems. It also describes how to troubleshoot DECmessageQ system problems.

System Configuration

Before you can develop and run DECmessageQ applications, you must define the run-time environment that is best for your application and system configuration. These procedures vary depending upon the environment in which you are running DECmessageQ software.

For all DECmessageQ environments, begin by defining the required number of message queuing buses, message queuing groups, and message queues. Depending on the number of buses, queues, and groups you configure, you must set up the system resources to ensure that sufficient resources are available to support application processing.

For example, in a UNIX environment, you must properly allocate shared memory, semaphores, and process slots, then rebuild the UNIX kernel. In an OpenVMS environment, you must properly allocate global memory, define OpenVMS symbols, and set system parameters.

For a complete description of how to configure DECmessageQ software, refer to the installation and configuration guide for the DECmessageQ product you are using.

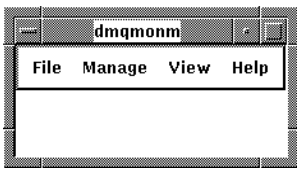
Monitoring System Performance

The DECmessageQ Monitor utility helps developers observe the DECmessageQ environment on local and remote nodes. Developers and system managers can use the summary and detailed display of information by the Monitor utility to tune the DECmessageQ system configuration. Though its display varies on UNIX, Windows NT, and OpenVMS systems, it is used to perform similar functions in all environments, such as:

- Shutting down a message queuing bus
- Starting a new message queuing group
- Shutting down a single message queuing group
- Starting a cross-group link
- Shutting down a cross-group link
- Collecting and displaying statistics for each queue
- Collecting and displaying statistics for each cross-group link

Figure 4–1 shows the main menu of the Motif based user interface (UI) Monitor utility available on UNIX systems.

Figure 4–1 Monitor Utility—Motif UI



To monitor or control your DECmessageQ groups or buses, select the activity you want to perform using the pull-down menus. In most cases, a secondary screen appears, prompting you to enter more information on the function that you want to perform.

For a complete description of how to use the Monitor utility, refer to the installation and configuration guide for the DECmessageQ product you are using.

Troubleshooting System Problems

DECmessageQ has an error logging mechanism to display and capture informational, warning, and error messages that can occur during processing. The messages display a description of the condition to help developers gather more information about failures in their DECmessageQ environment.

Log messages contain the following types of information:

- Warning messages, such as ‘‘Resources almost depleted’’
- Error notifications, such as ‘‘Message delivery failure’’
- Event notifications, such as ‘‘Reconnection from an external node’’

Messages and notifications can be logged into a file or sent to a terminal. For specific information on how to set up error logging, refer to the installation and configuration guide for the DECmessageQ product you are using.

Glossary

access control list (ACL)

A list that defines the kinds of access to be granted or denied to users of an object. Access control lists can be created or objects such as files and devices.

acknowledgment (ACK)

A status message that indicates the completion of an operation.

action routine

A subroutine that is executed when arrival of a message interrupts processing. When the action routine finishes, the original program resumes. See also *asynchronous system trap*.

active queue

A queue that is capable of receiving messages. Contrast with *inactive queue*.

address

See *queue address*.

allocate

To reserve a particular device for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

application

A program or collection of programs designed to perform a function.

application programming interface (API)

An interface used by application programs to call services external to the program. The API supports the exchange of information in a multivendor environment.

application protocols

An agreed set of rules that govern the management of connections between partner programs. See also *duplex connection* and *simplex connection*.

asynchronous

Pertaining to a style of message queuing whereby messages can be sent or received at any time without waiting for the receiver program to receive, process, or respond to a specific event. Contrast with *synchronous*.

asynchronous system trap (AST)

An software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously of the occurrence of a specific event. If a user has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. WHen the AST routine exits, the system resumes execution of the process at the point where it was interrupted.

attach

To make a process known to the DECmessageQ message queuing bus at a particular message queue.

attachment point

A particular queue location on the DECmessageQ message queuing bus that allows communication between processes without requiring a formal connection sequence.

blocking

Pertaining to a synchronous style of message delivery where the program is forced to wait for an action to complete. Contrast with *nonblocking*.

broadcast distribution

The action of delivering a message to all processes interested in a particular broadcast stream.

broadcasting

A style of communicating that uses one message sender program and multiple message receiver programs.

broadcast stream

A data message pipeline that has a single entry point and multiple exit points. Messages sent to the broadcast stream are simultaneously distributed to all registered queues. See also *private broadcast stream* and *universal broadcast stream*.

buffer

An internal memory area used for temporary storage of data records during input or output operations.

buffer pool

A common memory area that stores message buffers for a message queuing group. A buffer pool consists of fixed-size memory structures that can hold one message each.

bus ID

A reference value that distinguishes one DECmessage message queuing bus from another.

CISC

See *Complex Instruction Set Computer*.

class

A 16-bit piece of data that describes a grouping or category of message types. Also called *message class*. See also *type*.

client

A computing system entity that uses the services of other system entities called servers. See also *server*.

client/server model

A hardware or software system design used in developing distributed applications. In the client/server model, a server system provides common database access, performs computations, and assumes system management tasks for its clients.

Complex Instruction Set Computer

A computer that has individual instructions that perform complex operations, including complex operations performed directly on locations in memory.

COM Server

A server process that passes cross-group messages to other DECmessageQ message queuing groups. A COM Server creates the DECmessageQ message queuing bus environment and must be activated before message queuing can occur.

configuration editor

A Windows editor used for defining and managing DECmessageQ buses, groups, and related information.

configuration file

A text file comprised of information line items used to configure DECmessageQ software. The DECmessageQ LU6.2 Port Server uses a configuration file to configure itself during initialization and to determine the context in which it is operated.

confirmation

See *message confirmation*.

connectionless

Pertaining to not having a logical link. A connection does not have to be established with a partner process in order to pass information between them.

connection-oriented

Pertaining to a communication method where two partners must establish a connection before they can exchange messages.

cross-group

Pertaining to messages that pass between DECmessageQ message queuing groups. A cross-group message is targeted to a message queuing group outside of the local group. Cross-group connections enable applications to share information across different systems connected to the same message queuing bus.

datagram

A ''best effort'' style of message delivery in which a nonrecoverable attempt is made to deliver a message. If the message cannot be delivered to a target, then an error is logged.

dead letter journal (DLJ)

A file that provides nonvolatile disk storage for messages that cannot be stored for automatic recovery. Applications use the DLJ file to resend undelivered messages. Also called *DLJ file*.

dead letter queue (DLQ)

The permanent message queue that provides memory-based storage of all recoverable messages that could not be stored for automatic delivery. Also called *DLQ file*.

DECmessageQ message queuing bus

Digital's product implementation of the message queuing technology. The DECmessageQ message queuing bus is the highest level of user configuration. It provides a bus ID for specifying the logical communications data highway that an application uses for posting and retrieving messages.

delivery interest point

A component of the delivery mode that indicates the step in the message recovery data flow at which the sender program is notified.

delivery mode

A selection of options that specify how the sender program receives notification of recoverable message delivery and the point in the message flow at which the notification is sent. See also *message delivery*.

DQF file

See *destination queue file*.

distributed application

An application that divides the user interface, processing, or data among one or more units that execute on a single central processing unit (CPU) or multiple nodes in a network.

distributed computing

An application design methodology that places data entry and application processing close to departmental and functional end users. These users are most familiar with the input requirements and need the processed output to support their business objectives.

Distributed Name Services (DNS)

A facility that DECmessageQ uses to store global names. Using DNS on OpenVMS systems enables DECmessageQ to locate the queue address for a queue defined by any group on the message queuing bus. See also *naming*.

distribution

A stage in broadcast services where the SBS Server delivers a message to receiver programs.

distribution queue

A queue address that is specified in a broadcast or availability services registration message. The distribution queue is the final destination of a broadcast or availability notification message.

DLQ file

See *dead letter queue.*

duplex connection

An application protocol where the initiating partner is the sender program and the accepting partner is the receiver program, until the sender program requests a direction change and becomes the new receiver program. The accepting partner then becomes the sender program and remains the sender program until requesting a direction change. Contrast with *simplex connection*.

event

A network- or system-specific occurrence, such as timer expiration, for which the logging component maintains a record.

explicit confirmation

A type of message confirmation that requires the receiver program to delete the message from the recovery journal using a message sequence number. The message is not deleted until the receiver program has finished processing the information in it.

facility

A collection of one or more computer programs that implement a set of related functions or services. The implementation of a facility can consist of either a process or a procedure.

failover

1. The process of a reconfiguration after a hard fault or for planned maintenance.
2. The ability of a system or component to reconfigure itself.

full duplex

Pertaining to a communications method in which data can be transmitted and received at the same time.

global data structure

A data structure that can be shared by multiple processes.

global sections

An OpenVMS shared memory segment potentially available to all processes in the system. Access is protected by standard access control mechanisms.

group

See *message queuing group*.

group ID

The internal number of the DECmessageQ message queuing group. The group ID is part of the queue address. Each group ID must be unique within the message queuing bus.

group name

The symbolic name associated with the DECmessageQ group ID.

half-duplex

Pertaining to a communication method where one partner is sending data when the other partner is receiving data. See also *duplex connection*.

heterogeneous computing environment

An environment in which applications run on computer systems from different vendors employing various operating system and networking software.

heterogeneous messaging

The use of different communications methods to transfer messages.

heterogeneous operating systems

A configuration of a variety of computers and operating systems connected by networking hardware and software.

implicit confirmation

A type of message confirmation on DECmessageQ for UNIX and Windows NT systems that automatically deletes a recoverable message from a journal file. The receiver program does not need to respond to the receipt of the message.

inactive queue

A queue that is not capable of receiving messages. Contrast with *active queue*.

inbound conversation allocation

The allocation of conversations that are initiated by an OpenVMS transaction program.

inbound session

A generic term for a session used by LU6.2 Port Servers to initiate APPC/LU6.2 conversations with an IBM client. Contrast with *outbound session*.

inbound target

A name known to a DECmessageQ client that is used to establish a connection to an IBM client. See also *target*.

initiator-only deallocation

A method of duplex connection termination where the initiating partner is the only one who can terminate the connection normally. Contrast with *open deallocation*.

interprocess communication

Two-way communication between active independent processes.

journaled guaranteed delivery

A method used by applications to guarantee DECmessageQ message delivery in which the sending process sends a message that is delivered to the target disk queue.

journal file

A disk file that records all received and confirmed messages.

journaling

Writing to a journal file.

journal replay

A method for resending messages stored in the DLJ or PCJ files.

link driver

A process that establishes a communications link between message queuing groups. Using the queuing engine, each link driver sends outbound messages and delivers inbound messages.

Linked List Sections

A set of global sections that is used to store the DECmessageQ message buffers for a message queuing group. See *buffer pool*.

LU

Logical unit. For example, LU6.2 is the IBM networking protocol (SNA).

message

A data item that is transmitted over a communications medium. A message contains a message header and data portion. The message header is comprised of attributes, which are defined by the application program, and context, which is added by the messaging tool.

message-based services

Predefined request, notification, and response messages exchanged between the application and DECmessageQ server process.

message capture

A part of the Script Facility that provides a mechanism for viewing messages that are sent or received by an application.

message confirmation

An action taken by the receiver program, which indicates to the message queuing system that the processing of a recoverable message has completed. A message confirmation terminates the message system's responsibility for the recoverable message.

Message Control Section (MCS)

A global section that stores information about message queues and other global information, such as send and receive counters.

message delivery

The processing steps performed by the message queuing system when moving the message from a sender program to the receiver program's message queue.

message queue

An attachment point on the DECmessageQ message queuing bus where pending messages are stored. A message queue is identified by a queue number and can be primary, secondary, or multireader.

message queuing

A technique for sending messages from one program to another by directing messages to a queue. The queue stores the messages until they can be processed by the receiver program.

message queuing bus

A transparent communication mechanism that uses a simple logical bus topology. A message queuing bus provides a standard set of program-callable subroutines that allow message transfer between programs and message queues. See also *DECmessageQ message queuing bus*.

message queuing group

A set of logical addresses on the DECmessageQ message queuing bus, all sharing a set of common DECmessageQ resources. Each message queuing group resides on a single system. However, multiple groups can reside on the same system. The interconnections between groups define the extent of a message queuing bus.

Message Recovery Services (MRS)

A set of DECmessageQ services that manage the automatic redelivery of critical messages.

message queuing

Interprocess communication and information exchange between two or more cooperating processes accomplished by directing messages to a memory- or disk-based queue as an intermediate storage point.

modular

Pertaining to subdividing application code by organizing it in small, task-oriented segments.

module

A portion of a program or program library, as in a source module, object module, or image module.

Multipoint Outbound Target (MOT)

An entry point to a broadcast stream. A range of queue addresses is reserved to define a set of unique broadcast streams.

multireader queue (MRQ)

An optional queue type on the DECmessageQ message queuing bus that stores messages that can be read by several simultaneous readers. Each reader, in turn, receives the next message in first-in/first-out (FIFO) order from the queue. A multireader queue can be permanent or permanently active. See also *queue type*.

naming

Pertaining to the use of a symbolic entity in place of an actual value. DECmessageQ uses character strings for names, which, when translated, reveal queue addresses.

network

A collection of interconnected individual computer systems.

node

An individual computer system in a network that can communicate with other computer systems in the network.

nonblocking

Pertaining to an asynchronous style of message delivery where the program does not have to wait for an action to complete. The nonblocking style generally involves receiving an acknowledgment message when the action is complete. Contrast with *blocking*.

nontransparent

Relating to IBM systems, which use EBCDIC data encoding format. DECmessageQ clients expect data in ASCII format. For a target defined as nontransparent, the LU6.2 Port Server automatically provides bidirectional data encoding format translation between ASCII and EBCDIC data formats. Contrast with *transparent*.

notification

A type of message-based service that supplies up-to-date information on events as they occur.

open deallocation

A method of duplex connection termination where the current sender can terminate the connection normally, regardless of which partner initiated the connection. Contrast with *initiator-only deallocation*.

operand

Data in the message header or message data structure that will be compared.

OS

Operating system. An integrated collection of programs that controls the execution of computer programs and performs system functions.

outbound conversation allocation

The allocation of conversations that are initiated by a CICS transaction program.

outbound session

A generic term for a session used by an IBM client to initiate APPC/LU6.2 conversations with the LU6.2 Port Server. Contrast with *inbound session*.

outbound target

A name known to an IBM client that is used to establish a connection to a DECmessageQ client. See also *target*.

outbound transparent

A special LU definition that enables transparent data transmission from the IMS LU6.1 Adapter for LU6.2 Applications.

PAMS

Process Activation and Message Support. PAMS is the original name for the DECmessageQ message queuing system. The DECmessageQ API preserves the original product acronym in the name of each callable service to protect customer investment in application development.

PCJ file

See *postconfirmation journal file.*

pending

Pertaining to a message that is currently in a queue.

permanently active queue

A message queue that can receive messages without an application attachment. It retains its name and messages after the process detaches from DECmessageQ. See also *permanent queue*.

permanent outbound target

A type of outbound target that supports a method of message delivery where DECmessageQ clients can request that outbound traffic be delivered to a predetermined DECmessageQ queue. The queue must be a permanent queue in the designated group.

permanent queue

A message queue that is always at the same address on the DECmessageQ message queuing bus. It exists regardless of whether a process is attached to it. A permanent queue retains its name and address after the process detaches, but loses any pending messages. See also *permanently active queue.* Contrast with *temporary queue*.

platform

The combination of hardware, operating systems, and windowing systems that supports an application.

port server

A class of DECmessageQ application that provides a connection to the DECmessageQ message queuing bus for client applications executing on platforms that do not have a DECmessageQ implementation.

postconfirmation journal file (PCJ)

A disk file that holds confirmed recoverable messages that can be retrieved for audit trailing. Also called *PCJ file.*

primary queue

The one required queue used when a process attaches to the DECmessageQ message queuing bus. There can be only one primary queue for each process. It is used as the default return address on all messages sent by that process. A primary queue can be permanent, permanently active, or temporary. See also *queue type*.

private broadcast stream

A MOT address range indicating that messages are restricted to distribution by one SBS Server, which restricts distribution to queues that have registered with that SBS Server.

process

The basic entity scheduled by the system software, a process provides the context in which an image executes.

queue

See *message queue*.

queue address

A longword value that uniquely identifies the attachment point on the DECmessageQ message queuing bus. An address includes a group ID and a queue number.

queue attribute

A specific characteristic of a queue that determines the features of the queue. Some examples of queue attributes are: permanent or temporary, recoverable or volatile, FIFO or non-FIFO capability, and so on.

queue number

A number that represents a unique location of a permanent or temporary queue address within a DECmessageQ message queuing group. There must be at least one queue number for every process using the DECmessageQ message queuing bus.

queue type

A description of a message queue as being primary, secondary, or multireader.

queuing engine

A process that handles all message traffic between message queuing groups. One queuing engine is created for each group. The queuing engine creates the global sections of memory for message queues within the group.

quota

The total amount of a system resource, such as disk space, that a job is allowed to use in an accounting period.

receive message quota

The application-defined limitation (in bytes) on pending messages in a queue.

receiver program

The application program in a connection that is accepting messages from the sender program.

recoverable message

A message that is temporarily stored on a disk file and is guaranteed delivery if an application, system, or network fails.

Registry

A database in the Windows NT operating system that stores system and optional software configuration information.

reliable transmission

Pertaining to messages that are guaranteed to be delivered to a target queue. Contrast with *recoverable message*.

registration

A stage in broadcast services where an application program subscribes to a broadcast stream by sending a registration message to the SBS Server.

request

A type of message-based service that obtains information or registers to receive ongoing notifications.

response

A type of message-based service that provides information to fulfill requests or acknowledge registration and deregistration requests.

return-to-sender

A method of DECmessageQ message delivery in which a nonrecoverable attempt is made to deliver a message. If the message cannot be delivered, it is returned to the sending process marked with a special return status.

SAF file

See *store and forward.*

Script Facility

A productivity tool that speeds application testing by providing message simulation, capture, and replay abilities.

script file

A file with special syntax defining message information.

secondary queue

An optional private queue type used in conjunction with a primary queue. It provides a secondary address for messages. A secondary queue can be permanent, permanently active, or temporary. See also *queue type*.

Selective Broadcast Services (SBS)

DECmessageQ services that enable an application to send a message to many receiving applications with a single send operation.

semaphore

In DECmessageQ software, a common data structure used to serialize access to shared data structures.

sender notification

A component of the delivery mode that indicates how the sender program wants to receive information about the delivery of the message.

sender program

The application program in a connection that is sending messages to the receiver program.

server

A software module designed to perform a specific function for many users. See also *client* and *client/server model*.

sessionless

Pertaining to the absence of protocols required to manage communications between processes.

shared memory segment

A portion of memory that can be accessed by two or more processes.

simplex connection

An application protocol where the initiating partner is always the sender program and the accepting partner is always the receiver program. The receiver program can signal an error, but cannot send. Contrast with *duplex connection*.

SNA

An IBM networking protocol.

source queue

A queue address of the program that sent the message.

stateless

Pertaining to the absence of protocols to identify the stages within a message transmission.

store and forward (SAF)

A message recovery journal that provides nonvolatile storage on the sender's system for automatic recovery and delivery of messages. Also called *SAF file.*

submission

A stage in broadcast services where an application program inserts a message on a broadcast stream.

synchronous

Pertaining to a message queuing system where the sender program must wait for a specific event or reply. Contrast with *asynchronous*.

target

A generic term for the client application with which another client application wants to establish a connection.

target queue

The queue address of the receiver program of the message.

TCP/IP

Transport Control Protocol/Internet Protocol. TCP/IP is a set of protocols that governs the transport of information between computers and networks of dissimilar types. Both Internet and UNIX based systems use TCP/IP protocols.

temporary queue

A queue that exists only for the duration of the process attachment to the DECmessageQ message queuing bus. The assignment of the queue is not permanently defined. A temporary queue loses all messages in the queue when the process detaches from the queue. Contrast with *permanent queue*.

transparent

Relating to IBM systems, which use EBCDIC data encoding format. DECmessageQ clients expect data in ASCII format. For a target defined as transparent, the LU6.2 Port Server does not provide data encoding format translation. Contrast with *nontransparent*.

type

A 16-bit piece of data DECmessageQ uses to identify a kind of message from all other messages in the application. See also *class*.

undeliverable message action (UMA)

The action that occurs when the DECmessageQ message queuing bus is unable to store a message. The UMA specifies the action to be taken with the recoverable message if it cannot be stored for guaranteed delivery by the message recovery system.

universal broadcast stream

A MOT address range indicating that messages can be distributed by all SBS Servers. Distribution is across the entire message queuing bus wherever SBS software is running. See also *broadcast stream.*

user interface (UI)

The mode or style with which the system interacts with the user. A command line and windowing systems are user interfaces.

user process

A user's program image.

utility

A program that provides a set of related general-purpose functions, such as a program development utility (an editor, a linker).

wait for dequeue

A method of DECmessageQ message delivery in which the sending process is blocked until the message is read from the target queue by the receiver program.

wait for enqueue

A method of DECmessageQ message delivery in which the sender program process is blocked until the message is written to the target queue. A return status indicates if the message is successfully written to the queue. This delivery method guarantees message delivery when message recovery services are not available on the target platform.

Index

A

AIX support, 1–6
Alpha AXP support, 1–6
API, 3–1
Application programming interface
  See API
Application testing and debugging, 3–5
Asynchronous communication, 2–2
Attaching to the message queuing bus, 3–2

B

Broadcast stream, 2–7
Broadcasting messages, 2–6
Bus ID, 2–4

C

Client implementation, 1–7
Client/server computing, 1–2
Configuration requirements, 4–1

D

Dead letter journal file, 2–10
DECmessageQ
  message queuing bus, 2–3
DECmessageQ architecture, 2–3
DECmessageQ clients, 1–7
DECmessageQ message queuing bus, 1–4, 2–3
DECnet
  SNA Gateway support, 1–7
DECnet support, 1–6
Destination queue file, 2–10
Detaching from a queue, 3–3
Distributed computing, 1–1
DLJ, 2–10
DQF, 2–10

E

Error logging, 4–3

G

Group ID, 2–4

H

Heterogeneous computing environment, 1–5
HP-UX support, 1–6

L

LU6.2 protocol, 1–7

M

Macintosh support, 1–6
Message
  content, 2–1
  registering for broadcast, 2–7
Message broadcasting, 2–6

Message capture, 3–6
Message client, 1–7
Message confirmation, 2–10
Message delivery, 2–9
Message queue, 2–5
Message queuing, 1–4
  capabilities, 1–4
  datagram, 2–9
  definition, 2–1, 2–2
  how it works, 2–2
  journaled guaranteed delivery, 2–9
  priority, 2–8
  purpose and use, 2–1
  return to sender, 2–9
  store and forward, 2–9
  wait for dequeue, 2–9
  wait for enqueue, 2–9
Message queuing bus, 1–4, 2–3
  attaching to, 3–2
  configuration, 2–4
Message queuing group, 2–4
Message queuing system, 2–1
Message recovery, 2–9
  files, 2–10
  replaying journal files, 2–11
Message Recovery Services (MRS), 2–10
Message replay, 3–6
Message selection, 2–8
Message server, 1–5
Message simulation, 3–5
MONITOR utility, 4–2
Monitoring performance, 4–2
MS–DOS support, 1–6
Multipoint Outbound Target (MOT), 2–7
Multireader queue, 2–5
Multivendor support, 1–5
  API services, 3–4
  configuration, 4–1
  delivery options, 2–11
  message broadcasting, 2–8
  queue types, 2–5
  script facility, 3–6

O

88open support, 1–6
OpenVMS AXP support, 1–6
OpenVMS VAX support, 1–6
OS/2 support, 1–6
OSF/1 AXP support, 1–6

P

pams_attach_q, 3–2
pams_cancel_get, 3–4
pams_cancel_timer, 3–3
pams_confirm_msg, 3–4
pams_exit, 3–3
pams_get_msg, 3–2
pams_get_msga, 3–4
pams_get_msgw, 3–3
pams_locate_q, 3–3
pams_put_msg, 3–2
pams_set_timer, 3–3
PCJ, 2–10
Peer-to-peer communication, 2–2
Postconfirmation journal file, 2–10
Primary queue, 2–5
Priority messages, 2–8
putil_show_pending, 3–4

Q

Queue
  attaching to, 2–3
  detaching from, 3–3
Queue address, 2–5
Queue number, 2–5
Queue type, 2–5

R

Receiving messages, 2–2, 3–2
Recovering messages, 2–9
Registration message, 2–7

S

SAF, 2–10
SBS Server, 2–7
Script facility, 3–5
Script file, 3–5
Scripting language, 3–5
Secondary queue, 2–5
Selective broadcast services (SBS), 2–6
Sending messages, 2–2, 3–2
Solaris support, 1–6
Store and forward file, 2–10
SunOS support, 1–6
Synchronous communication, 2–2
System configuration, 4–1
System management, 4–1
System performance
  monitoring, 4–2

T

TCP/IP support, 1–6
Timers
  using, 3–3
Troubleshooting problems, 4–3

U

ULTRIX RISC support, 1–6
ULTRIX VAX support, 1–6

W

Windows NT support, 1–6
Windows support, 1–6