SFA Modernization Partner

United States Department of Education

Student Financial Assistance

EAI Core Architecture

EAI Application Enablement Guide

Release One

Task Order #54

Deliverable # 54.1.4

July 13, 2001

TABLE OF CONTENTS

1. INTRODUCTION 5

1.1 PURPOSE 5

1.2 SCOPE 5

1.3 DOCUMENT ORGANIZATION 5

2. MQSERIES ARCHITECTURE GUIDELINES 7

2.1 NAMING GUIDELINES 7

2.1.1 Common Rules 7
2.1.2 Queue Manager 8
2.1.3 Local Queues 9
2.1.4 Remote Queues 10
2.1.5 Alias Queues 11
2.1.6 Model and Dynamic Queues 11
2.1.7 Transmission Queues 12
2.1.8 Dead Letter Queues 13
2.1.9 Initiation Queues 14
2.1.10 Processes 14
2.1.11 Channels 15

2.2 USING A MQSERIES OBJECT 15

2.2.1 Channels 15
2.2.2 Queues 15

2.3 MQSERIES MESSAGING IMPLEMENTATION GUIDELINES 17

2.4 MQSERIES CLUSTER DESIGN GUIDELINES 17

2.4.1 Selecting Queue Managers to Hold Repositories 17
2.4.2 Organizing a cluster 18
2.4.3 Overlapping clusters 19
2.4.4 In the Unlikely Event of a Repository Failure 20
2.4.5 Cluster channels 21

2.5 MQSERIES CLUSTER IMPLEMENTATION GUIDELINES 21

2.6 SFA CLUSTER SPECIFICS 22

2.6.1 Physical layout of the cluster 22
2.6.2 Cluster configuration 23

2.7 MQSERIES WEBSPHERE DESIGN GUIDELINES 24

2.7.1 WebSphere Connectors 24
2.7.2 Architecture look and feel 26
2.7.3 SFA EAI WebSphere Reusable Component 27

3. APPLICATION INTEGRATION 28

3.1 LEGACY SYSTEM APPLICATION EAI INTEGRATION CONSIDERATIONS 28

3.2 CPS 28

3.3 NSLDS 29

3.3.1 NSLDS Batch 30
3.3.2 NSLDS Transaction 32

3.4 DLSS 32

3.5 PEPS 33

3.6 BTRADE 33

4. SFA APPLICATION ENABLEMENT GUIDELINES 35

4.1 APPLICATION PROGRAMS AND MESSAGING 35

4.2 APPLICATION USAGE GUIDELINES 35

4.2.1 Identifying an Application for a Queue Manager 36
4.2.2 Opening and Closing Queues 37
4.2.3 Putting Messages On A Queue 38
4.2.4 Getting Messages From A Queue 38
4.2.5 Queue Manager Connectivity Guidelines 39
4.2.6 Connecting To and Disconnecting From a Queue Manager 39
4.2.7 Pass the Connection Name as a Program Parameter 40
4.2.8 Messaging Using More Than One Queue Manager 40

4.3 APPLICATION INTERFACE PROGRAMMING OPTIONS 41

4.3.1 Message Delivery 41
4.3.2 Message Content 42

4.4 EAI COMMON ERROR HANDLING GUIDELINES 44

4.4.1 Failure of a MQI Call 44
4.4.2 System Interruption 44
4.4.3 Unable to Process Messages 44
4.4.4 Responding to Errors 45

4.5 TRIGGERED QUEUES AND APPLICATIONS 45

4.5.1 Designing MQSeries Applications 45
4.5.2 Starting MQSeries Applications 46

5. APPLICATION CONNECTIVITY (ADAPTERS AND BRIDGES) 48

5.1 MQSERIES APPLICATION ADAPTER 48

5.2 ADAPTER CLASSIFICATIONS 48

5.2.1 Type of Message 48
5.2.2 Interface Type 48

5.3 MQSERIES-CICS/ESA BRIDGE 49

5.3.1 Using the CICS Bridge 49

5.3.2 CICS Bridge at Work *49*

5.4 RUNNING CICS DPL PROGRAMS 50

5.4.1 Running CICS 3270 transactions *51*

6. COMMITTING AND BACKING OUT UNITS OF WORK 53

6.1 COMMITTING AND BACKING OUT 53

6.2 SYNCPOINT COORDINATION, SYNCPOINT, UNIT OF WORK 53

6.3 SINGLE-PHASE COMMIT 53

6.4 TWO-PHASE COMMIT 53

7. SYNCPOINT GUIDELINES 55

7.1 SYNCPOINTS IN MQSERIES FOR WINDOWS NT, MQSERIES ON UNIX SYSTEMS 55

7.1.1 Local units of work *55*

7.1.2 Global units of work *55*

7.1.3 Internal syncpoint coordination *56*

7.1.4 External syncpoint coordination *56*

7.1.5 Interfaces to external syncpoint managers *57*

7.2 MQSERIES SYNCPOINT CALLS FOR OS/390 58

7.3 MQSERIES SYNCPOINT CALLS ON WINDOWS NT AND UNIX SYSTEMS 58

8. NEXT STEPS 60

9. APPENDIX A: REFERENCE MATERIAL 61

10. APPENDIX B: GLOSSARY 63

1. INTRODUCTION

1.1 Purpose

The EAI Application Enablement Guide was developed in support of the Department of Education's Student Financial Assistance (SFA) Modernization Program, to provide an overview of the MQSeries Messaging functionality being implemented as part of the Enterprise Application Integration (EAI) project. The EAI provides a standard reusable architecture for connecting disparate, heterogeneous systems through common middleware architecture. The EAI architecture is built using MQSeries Messaging and MQSeries Integrator. This deliverable defines the guidelines for enabling SFA application developers to design and implement applications utilizing the features of the EAI Core architecture, as defined for the initial release.

The deliverable also serves to provide a high level overview of the features and capabilities of the SFA EAI Messaging infrastructure architecture and product capabilities. This deliverable should be the initial reading for all application developers who will be developing applications to utilize the EAI Bus at SFA. In addition, this deliverable will document the Release One EAI Core Architecture for the legacy systems MQ Adapters and their applicability for use in connecting to each Release One legacy system.

The document is intended to be a living document and a repository of MQSeries best practices and guidelines, which can be adopted by SFA for the implementation of EAI applications.

1.2 Scope

The EAI is a strategic component of the overall SFA enterprise architecture. The scope of this deliverable is to provide guidelines and best practices for designing and implementing applications which will utilize the services provided by the EAI core architecture. The guidelines defined in this deliverable are based on best practices and provide a structured approach for defining a consistent and maintainable environment to provide a framework for application developers to have consistent look and feel in developing EAI components. The deliverable also provides a set of reusable components for developing applications to utilize the features of the EAI Bus on each Release One legacy system.

1.3 Document Organization

This deliverable is divided into the following sections:

- Section 1 – Introduction

  This section provides an introduction and overview of the EAI Enablement Guide.

- Section 2 – MQSeries Architecture Conventions and Guidelines

  This section will provide guidance on naming conventions for using MQSeries in the SFA EAI architecture. These guidelines provide guidance in defining and implementing MQSeries objects.

- Section 3 – Developing SFA Applications to Utilize the EAI Core Architecture

  This section will provide guidance on naming conventions for using MQSeries in the SFA EAI architecture. These guidelines provide guidance in defining and implementing MQSeries objects.

- Section 4 – SFA Application Enablement Guidelines

This section provides an overview of messaging and provides specifics steps an application needs to perform in order to connect to a queue manager and to send and receive messages.

- Section 5 – Application Connectivity (Adapters and Bridges)

  This section discusses the use of adapters and bridges. Adapters handle data inbound-to and outbound-from the application or environment and a bridge is a piece of software that moves data between a message on a queue and an application or environment.

- Section 6 – Committing and Backing Out Units of Work

  This section describes how to commit and back out any recoverable get and put operations.

- Section 7 – Syncpoint Guidelines

  This section describes applications and their use of operating under syncpoint control.

- Section 8 – Next Steps

  This section provides a few ideas on how to continue to enhance the effectiveness of the EAI project.

- Section 9 – Appendix A: Reference Material

  This section provides the URL links to on-line documentation referenced throughout this document.

- Section 10 – Appendix B Glossary

  This section provides a glossary of MQSeries related terms and abbreviations.

2. MQSERIES ARCHITECTURE GUIDELINES

This section will provide guidance on naming conventions for using MQSeries in the SFA EAI architecture. These guidelines provide guidance in defining and implementing MQSeries objects.

2.1 Naming Guidelines

This section defines MQSeries Messaging naming guidelines for MQSeries objects within SFA's enterprise technical architecture.

2.1.1 Common Rules

All MQSeries names should follow MQSeries naming conventions, rather than the standard for object names on each supported platform. Key standards and guidelines:

- MQSeries object names should be created with all upper case letters.
  MQSeries allows both uppercase and lowercase letters in its names. However, MQSeries names are case sensitive. Using lowercase and uppercase characters for object names is a common source for naming errors.

- MQSeries object names should not include the ' % '.
  Although the '%' is a valid character, it is recommended that it not be used.

- MQSeries object names should be limited to alpha – numeric characters
  The underscore character (_), the forward slash character (/) and the period (.) can be used as separation characters in the name of an object. For example, "dlssloanappl" is not very legible. Instead, if the name contains a period it becomes "dlss.loan.appl", which can be considered more legible.

  These standards recommend using hierarchical names under certain conditions. One such example is to use a suffix where there are multiple "versions" of an object. For example, if you have a DLSS development queue and a DLSS production queue residing on the same system, the queue names could be created as: DLSS.DEV.FROM.EAI and DLSS.PRD.FROM.EAI.

- MQSeries object names should include a description. All objects have a DESCR attribute for this purpose. MQSeries does not act on the value, but it provides additional information as to the function of the queue.

  - MQSeries definitions should be saved. It is recommended that the MQSeries object definitions include a brief statement documenting for what the object is used. This documentation can reside anywhere; however, it is most useful to include it in the script file containing the MQSeries object definitions. It is recommended that on each MQSeries system a file be created at the root level of the mqm account to hold the MQSeries definitions. Any changes to the MQSeries object definitions should be done in the script file. The script file would be executed to define the MQSeries objects.

  Definitions should be saved for the following reasons:

  - In the case of a system failure, objects may need to be recreated. To perform this function, the definitions need to be saved separately from the queue manager.

- The script file can be used to reset the attributes to a known state. For example if triggering has been turned off, or GET or PUT disabled, it is helpful to be able to restore the objects to their initial state.
- The definitions supplement the MQSeries documentation.

2.1.2 Queue Manager

A queue manager provides the messaging and queuing services to application programs through Message Queue Interface (MQI) program calls. . A queue manager can be understood as a "container" for queues and related objects. There is typically one per system, but additional queue managers can be defined.

The following guidelines should be followed when naming queue managers:

- Assign unique names to all queue managers.
- If ReplyToQMgr is left blank in the Message Descriptor, MQSeries inserts the actual local Queue Manager name, not its alias.
  - Dead Letter Queue messages identify the real Queue Manager, not any alias.
- When defining objects, refrain from copying documentation examples. Copying the documentation examples provided with the installation files is an easy way to produce queue managers with duplicate names. Plan for the names of queue managers ahead of time.
- MQSeries queue manager names should be short and descriptive.
- A recommendation would be to make queue manager names the same as the network host name. However, keep the following points in mind:
  - On MVS, the queue manager name corresponds to the MVS subsystem name. Therefore, the queue manager name is restricted to four characters.
  - Many queue managers use the first eight characters when generating unique message identifiers.
  - Channel names, which by convention are derived from queue manager names, are limited to 20 characters.
  - If there were no obvious name, most users would adopt a convention for constructing a queue manager name. Make sure that the convention provides for further expansion, particularly where the restricted names on MVS are concerned.
- When naming queue manager aliases use the naming conventions for the specific platform. This feature is usually related to defining multiple channels between a pair of queue managers.

Naming conventions specific to SFA are discussed in the following sections.

Naming Convention for MQSeries Queue Manager for Mainframe (CPS and NSLDS on OS/390)

Some naming examples for MQSeries queue managers on the OS/390 are illustrated below. A numeric identifier may be appropriate where a processor (or hardware cluster) has multiple queue managers.

Example:

CST1

C – System

"C" for CPS and "N" for NSLDS

S – LPAR

"S" for System, "T" for Test and "P" for Production

T – Type of Regions supported

"T" for Test and Development and "P" for Production

1 – Occurrence of Queue Manager

A sequential number of 1 through 9

Naming convention of the MQSeries Queue Managers for all other platforms

The following queue manager names were used in defining the Release One EAI Core architecture based on the node names of the servers. This was done to be consistent with the naming of the network nodes. It will also allow a system administrator a better understanding of where a particular MQSeries queue manager is within the network without keeping an extra spreadsheet to define MQSeries queue managers and the boxes on which the queue managers reside.

Examples:

SU35E5	-	WAS Queue manager
SU35E16	-	EAI Bus Queue Manager on the SU35E16 EAI Server

2.1.3 Local Queues

A local queue object defines a local queue belonging to the queue manager to which applications are connected. The following guidelines should be adhered to when naming local queues:

- Local queue names can be up to 48 characters long. The queue names should be short, but long enough to be descriptive.
- Local queue names should not include the name of the queue manager or an indication of the platform used.
- Local queue names should not indicate that the queue is local.
- Local queue names should not include the words local or queue.
- Local queue names should be 48 characters or less in length and of the form:

  FIRSTNODE.SECONDNODE.THIRDNODE.FOURTHNODE

  where

  - FIRSTNODE represents the name of the system that owns the object. FIRSTNODE is replaced with characters indicating the name of the system that owns the object. This will be useful when applications from multiple business units share the same machine/queue manager.
  - SECONDNODE represents the direction the message is flowing. SECONDNODE is replaced with the literal TO or the literal FROM, indicating the direction of the queue.
  - THIRDNODE represents the system in which the message is going to or coming from. The THIRDNODE is replaced with characters defining which system the queue is going to or coming from.

- FOURTHNODE represents the application or function that is being performed by the queue. The FOURTHNODE is replaced with any number of characters, such that entire queue name does not exceed 48 characters in length, that is a unique and descriptive term for the application or business-specific function performed by the queue.

Examples:

NSLDS.FROM.EAI.REQPELL

EAI.FROM.WAS.REQPELL

2.1.4 Remote Queues

A remote queue object identifies a queue belonging to another queue manager. The remote queue is usually given a local definition. The definition specifies the name of the remote queue manager where the queue exists as well as the name of the remote queue itself. The information specified when defining a remote queue object enables the queue manager to find the remote queue manager, so that any messages destined for the remote queue go to the correct queue manager. The following guidelines should be adhered to when naming remote queues:

- Remote queue names can be up to 48 characters long. The remote queue names should be short, but long enough to be descriptive.
- Remote queue names should not include the name of the queue manager or an indication of the platform used.
- Remote queue names should not indicate that the queue is remote.
- Remote queue names should not include the words remote or queue.
- Remote queue names can be up to 48 characters in length and should be of the form:

  FIRSTNODE.SECONDNODE.THIRDNODE.FOURTHNODE

  - FIRSTNODE represents the name of the system that owns the object. FIRSTNODE is replaced with characters indicating the name of the system that owns the object. This will be useful when applications from multiple business units share the same machine/queue manager.
  - SECONDNODE represents the direction the message is flowing. SECONDNODE is replaced with the literal TO or the literal FROM, indicating the direction of the queue.
  - THIRDNODE represents the system in which the message is going to or coming from. The THIRDNODE is replaced with characters defining which system the queue is going to or coming from.
  - FOURTHNODE represents the application or function that is being performed by the queue. The FOURTHNODE is replaced with any number of characters, such that entire queue name does not exceed 48 characters in length, that is a unique and descriptive term for the application or business-specific function performed by the queue.

Examples:

EAI.TO.NSLDS.REQPELL

WAS.FROM.EAI.REPLYPELL

2.1.5 Alias Queues

An alias queue object enables applications to access queues by referring to them indirectly in MQI calls. When an alias queue name is used in a MQI call the name is resolved to the name of a message queue at run time. This enables changes to the queues that applications use without changing the application itself in any way. The following guidelines should be adhered to when naming alias queues:

- Alias queue names can be up to 48 characters long. The alias queue names should be short, but long enough to be descriptive.
- Alias queue names should not include the name of the queue manager or an indication of the platform used.
- Alias queue names should not indicate that the queue is an alias.
- Alias queue names should not include the words alias or queue.
- Alias queue names can be of the form:

  FIRSTNODE.SECONDNODE.THIRDNODE.FOURTHNODE

  - FIRSTNODE represents the name of the system that owns the object. FIRSTNODE is replaced with characters indicating the name of the system that owns the object. This will be useful when applications from multiple business units share the same machine/queue manager.
  - SECONDNODE represents the direction the message is flowing. SECONDNODE is replaced with the literal TO or the literal FROM, indicating the direction of the queue.
  - THIRDNODE represents the system in which the message is going to or coming from. The THIRDNODE is replaced with characters defining which system the queue is going to or coming from.
  - FOURTHNODE represents the application or function that is being performed by the queue. The FOURTHNODE is replaced with any number of characters, such that entire queue name does not exceed 48 characters in length, that is a unique and descriptive term for the application or business-specific function performed by the queue.

Examples:

EAI.TO.NSLDS.REQPELL

WAS.FROM.EAI.REPLYPELL

Note: Alias queues were not used by the Release One - EAI Core team

2.1.6 Model and Dynamic Queues

The model queue object defines a set of queue attributes that are used as a template for a dynamic queue. The queue manager creates dynamic queues when an application makes an open queue request specifying a model queue. The dynamic queue that is created in this way is a local queue whose name is specified by the application and whose attributes are the same as the model queue.

Model Queue Naming Conventions

Model queues were not used by the Release One - EAI Core team. If model queues are required, the following design guidelines are applicable. Model queue names can be up to 48 characters in length and should be of the form:

FIRSTNODE.SECONDNODE.THIRDNODE.FOURTHNODE

- FIRSTNODE represents the name of the system that owns the object. FIRSTNODE is replaced with characters indicating the name of the system that owns the object. This will be useful when applications from multiple business units share the same machine/queue manager.
- SECONDNODE represents the direction the message is flowing. SECONDNODE is replaced with the literal TO or the literal FROM, indicating the direction of the queue.
- THIRDNODE represents the system in which the message is going to or coming from. The THIRDNODE is replaced with characters defining which system the queue is going to or coming from.
- FOURTHNODE represents the application or function that is being performed by the queue. The FOURTHNODE is replaced with any number of characters, such that entire queue name does not exceed 48 characters in length, that is a unique and descriptive term for the application or business-specific function performed by the queue.

Examples:

EAI.TO.NSLDS.REQPELL

WAS.FROM.EAI.REPLYPELL

Note: Model queues were not used by the Release One - EAI Core team

2.1.6.1 Dynamic Queue Naming Conventions

Dynamic queues were not used by the Release One - EAI Core team. If dynamic queues are required, the following design guidelines are applicable. Dynamic queue names can be up to 48 characters in length and should be of the form:

FIRSTNODE.SECONDNODE.THIRDNODE.FOURTHNODE.*

- FIRSTNODE represents the application and can be up to four characters in length.
- SECONDNODE represents the *AppName* which is up to eight characters in length. SECONDNODE should be replaced with the name of the application using this queue.
- THIRDNODE should be replace with the literal DYNQ indicating this is a dynamic queue.
- FOURTHNODE represents the application ID and is up to eight characters in length.
- The Asterisk (*) must follow the prefix name, so MQSeries may generate a unique name with the remaining characters. Note that the prefix name can only be up to 32 characters, so MQSeries can create the unique name at the end.

  Example:

  ACCT.LOANS.DYNQ.PENDING.*

  Note: Dynamic queues were not used by the Release One - EAI Core team

2.1.7 Transmission Queues

A transmission queue temporarily stores messages that are destined for a remote queue manager. Transmission queues must be defined for each remote queue manager that a local queue manager will send messages to. It is possible to associate several transmission queues with different characteristics to remote queue manager. This allows different classes of transmission service. Transmission queue names can be 48 characters or less in length. The following guidelines should be adhered to when naming transmission queues:

- Transmission queue names will include the name of the adjacent (i.e. directly connected) queue manager. The transmission queue name will be the name of the destination queue manager only in the case where the destination queue manager is directly connected with the sending queue manager.
- When the destination queue manager is not directly adjacent to the sending queue manager the transmission queue name should be the name of the queue manager that is directly adjacent to the sending queue manager.
- If there is only one channel to the queue manager, use the exact name of the adjacent queue manager.
- If there will be multiple channels to the queue manager, use the adjacent queue manager name followed by a dot and some distinguishing characteristic. The distinguishing characteristic could be a number, a letter, or anything else to designate one channel from another.
- If the exact queue manager name is not used, appropriate queue manager alias definitions need to be provided to allow MQSeries to perform queue manager name resolution.
- Transmission queue names should be of the form:

  FIRSTNODE[.SECONDNODE]

  Where:

  - FIRSTNODE represents the adjacent queue manager.
  - SECONDNODE represents the distinquishing characterstic to uniquely identify the transmission queue to the queue manager.

  Examples:

  SU35E5.1

  SU35E5.2

  CPT1

2.1.8 Dead Letter Queues

A dead-letter queue, also known as an undelivered-message queue receives messages that cannot be routed to their correct destinations. This will occur when, for example:

- The destination queue is full.
- The message cannot be put on the destination queue.
- The sender is not authorized to use the destination queue.
- The destination queue does not exist.

Dead Letter queue names should be 48 characters or less in length and should adhere to the following guideline:

FIRSTNODE.SECONDNODE.THIRDNODE.FOURTHNODE

Where:

- FIRSTNODE represents the queue manager name.
- The SECONDNODE should be replaced with literal DEAD.
- The THIRDNODE should be replaced with the literal LETTER.

– The FOURTHNODE should be replaced with the literal QUEUE.

Example: SU35E16.DEAD.LETTER.QUEUE

2.1.9 Initiation Queues

An initiation queue receives trigger messages, which indicate that a trigger event has occurred. A trigger event is caused by a message that satisfies the specified conditions being put onto a queue. Messages are read from the initiation queue by a trigger monitor application. The trigger monitor application starts the appropriate application to process the message. If triggers are active, at least one initiation queue must be defined for each queue manager. Initiation queue names should be 48 characters or less in length and of the form:

FIRSTNODE.[SECONDNODE.]THIRDNODE.

- FIRSTNODE represents the system name and should be replaced with the name of the system.
- SECONDNODE is optional and could represent the type of function being performed by the system.
- THIRDNODE should be replaced with the literal INIT or the literal INITQ, literal standing for the initiation queue.

Example:

CPS.BATCH.INIT

DLSS.INIT

2.1.10 Processes

A process definition object defines an application to a MQSeries queue manager. Typically in MQSeries, an application puts or gets messages from one or more queues and processes them. A process definition object is used for defining applications to be started by a trigger monitor. The definition includes the application ID, the application type, and application specific data. The following guidelines should be adhered to when naming processes:

- Process names should not include the name of the queue manager or an indication of the platform used.
- All process names can be up to 48 characters in length and should be of the form:

  FIRSTNODE.SECONDNODE.THIRDNODE

  Where:

  - FIRSTNODE is the name of the system.
  - SECONDNODE represents the application name or business function name and should be replaced with such.
  - THIRDNODE is the literal PROCESS indicating this MQSeries object is a process definition.

  Examples:

  NSLDS.PELL.PROCESS

  CPS.BATCH.PROCESS

2.1.11 Channels

A channel provides a communication path. There are two types of channels, message channels and MQI channels. A message channel provides a communication path between two queue managers on the same, or different, platforms. The message channel is used for the transmission of messages from one queue manager to another, and also shields the application programs from the complexities of the underlying networking protocols. A message channel can transmit messages in only one direction. If two-way communication is required between two queue managers, two message channels are required.

A MQI channel connects a MQSeries client to a queue manager on a server machine. It is for the transfer of MQI calls and responses only and is bi-directional. A channel definition exists for each end of the link. The following guidelines should be adhered to when naming both message channels and MQI channels:

- Channel names can be up to 20 characters long.
- Channel names should be of the form:

  FIRSTNODE.SECONDNODE.[THIRDNODE]

  Where:

  - FIRSTNODE represents the sending queue manager and should be replaced with the sending queue manager name.
  - SECONDNODE represents the receiving queue manager and should be replaced with the receiving queue manager name.
  - THIRDNODE is optional and represents an identifier used to distinguish the channel from other channels. THIRDNODE is only necessary if there exists more than one channel with the same name .

  Based on the above channel-naming convention, channel names can always be interpreted as *FromQueueManager.ToQueueManager* without ambiguity.

  Examples:

  SU35E16.NTT1.A

  NTT1.SU35E16

2.2 Using a MQSeries Object

This section documents when to use each one of the MQSeries objects covered in section 2.1.

2.2.1 Channels

In order for two machines to communicate via MQSeries, a channel must exist. If two systems must exchange messages, then two channels are required. Channels are created by system administrators or dynamically by the MQSeries queue manager. Although used by the MQSeries queue manager to move messages from one system to another, channels are of little interest to the application developer.

2.2.2 Queues

MQSeries system queues are simple FIFO disk-resident buffers that hold messages. Queues can be divided into local queues and remote queues. Local queues reside on the local system and remote queues reside on a remote system. If messages are destined for a remote system, then a remote queue should be used. Messages destined for applications on the local system are sometimes referred to as destination queues, application queues, or as local queues. Local queues are usually looked upon as queues from which applications GET messages. Queues should be created based on application needs and used when

messages need to move between systems or between applications on the same system. Local queues were used on each SFA legacy system.

Another type of queue is a transmission queue. Messages destined for remote queue managers are placed in special queues called transmission queues. Messages reside in the transmission queue until they can be delivered to the remote system via the sender channel. From the perspective of the local system, transmission queues hold outbound messages. Again, transmission queues are created by the system administrator and could be considered background objects. Transmission queues are used when messages are PUT to a remote queue; they are not written to directly by the application developer. At least one transmission queue must be defined for each remote queue manager to whom the local queue manager is to send messages directly. Transmission queues were used on each SFA legacy system.

Remote queues and alias queues are alternative logical names, which can be used to address a MQSeries system queue instead of using the actual queue name. In the case of the remote queue definition, a single name is provided for use by an application that relieves the application of needing to know the location (queue manager name) of the destination queue. Remote queues are used when sending messages to a destination queue defined on a remote queue manager. Both remote queues and alias queues are used by the application developer to get and put messages. Remote queues were used on each SFA legacy system.

Alias queues provide a simple one-to-one name substitution capability. An alias associates an alternative (alias) name with an already defined queue. By defining an alias, the MQSeries system administrator has the ability to redirect message traffic. By using alias queue definitions, the programmer is insulated from changing their application code to fit the changing needs of the network. An alias queue is not a queue, but an object that one can use to access another queue.

Initiation queues are queues that are used in triggering. A queue manager puts a trigger message on an initiation queue when a trigger event occurs. A trigger event is a logical combination of conditions that is detected by a queue manager. Initiation queues are defined by the system administrator for the use of triggering. Initiation queues are not used for the get and put of messages by the application developer. They are used by the queue manager. Initiation queues were defined and used on each SFA legacy system.

A dead letter queue is a queue that stores messages that cannot be routed to their correct destinations. There should only be one dead letter queue defined on each queue manger. The dead letter queue is defined by the system administrator at the time the queue manager is created. Applications can also use the queue for messages they cannot deliver. Dead letter queues were created on each SFA legacy system.

A model queue defines a set of queue attributes that are used as a template for creating a dynamic queue. Dynamic queues are created by the queue manager when an application issues a MQOPEN request specifying a queue name that is the name of a model queue. The dynamic queue that is created in this way is a local queue whose attributes are taken from the model queue definition. Dynamic queues do not survive product restarts; use dynamic queues with caution. Model and dynamic queues are used based on application needs. These were not used for SFA

Processes allow an application to be started without the need for operator intervention. An application queue can have a process definition object associated with it that holds details of the application that will get messages from the application queue. Processes are usually associated with a trigger event: when the

trigger event conditions are met, the application associated with the process is initiated. For SFA, processes were used to start the adapters.

2.3 MQSeries Messaging Implementation Guidelines

The following is a list of suggestions for MQSeries design and administration:

- The MQSeries Administrator is responsible for defining and maintaining MQSeries objects such as queues, queue managers, channels, and processes.
- The configuration values of MQSeries objects should be selected carefully to satisfy the requirements of each application. The default value is usually the recommended value. It should not be changed without careful evaluation.
- Include a Dead Letter Queue for every implementation.
- Avoid trigger types “DEPTH” and “EVERY”. These triggering methods have the potential to overload the system.
- Long running units of work are detrimental to the performance of the network. Break the work into small pieces; this tends to have the additional benefit of improved restart capability.
- Use verified network port addresses. Every queue manager needs a listener port in order to negotiate communications and manage the various queues. The default port address is 1414. Check with the network engineers to avoid any port address conflicts during implementation.
- Always evaluate using clusters of queues for redundancy and load balancing.
  Clusters provide a means to distribute the work in a queue among multiple processes. These processes may be on the same or different physical machines, and the machines may be located in the same or different locations. The only restriction on the locations of the members is that the members must be able to communicate via TCP/IP. Communications between the queue managers participating in each cluster enable the sending queue manager to route the message to the appropriate queue manager based on the default load balancing method or user defined cluster workload exit routine

2.4 MQSeries Cluster Design Guidelines

2.4.1 Selecting Queue Managers to Hold Repositories

In each cluster, select at least one, preferably two, or possibly more of the queue managers to hold repositories. A cluster could work quite adequately with only one repository but using two improves availability. The repository queue managers are interconnected by defining cluster-sender channels between them. A repository is a collection of information about queue managers that are members of a cluster. This information includes queue manager names, their locations, their channels, what queues they host, and so on. Typically, two queue managers in a cluster hold a full repository. The other queue managers in a cluster inquire on the information in the full repositories and build up their own subsets of this information in partial repositories.

The hardware architecture implemented at SFA can be seen in the diagram below. The cluster is configured to include the Websphere Application Server and the two Sun Solaris Servers. The Sun Servers were selected to be the repositories for the cluster.

- The most important consideration is that the queue managers chosen to hold repositories need to be reliable and well managed.

- Consider the location of the queue managers and choose ones that are in a central position geographically or perhaps ones that are located on the same system as a number of other queue managers in the cluster.

- Another consideration might be whether a queue manager already holds the repositories for other clusters. If a queue manager is a repository for one cluster, it would be wise to use the same queue manager as a repository for other clusters of which it is a member.

When a queue manager sends out some information about itself, or requests some information about another queue manager, the information or request is sent to two or more repositories. A repository handles the request whenever possible but if the chosen repository is not available another repository is used. When the first repository becomes available again, it collects the latest new and changed information from the others so that the queue managers are kept in synch. The repository queue managers send messages to each other to be sure that they are both kept up to date with new information about the cluster. The automatic updating of repositories by queue managers is part of the behavior that is inherent to clusters and is done behind the scenes without any intervention by the user.

The following cluster-sender and cluster-receiver definitions were taken directly from the IBM MQSeries Queue Manager Clusters Manual:

"A cluster-sender (CLUSSDR) channel definition defines the sending end of a channel on which a cluster queue manager can send cluster information to one of the full repositories. The cluster-sender channel is used to notify the repository of any changes to the queue manager's status, for example the addition or removal of a queue. It is also used to transmit messages. The repository queue managers themselves have cluster-sender channels that point to each other. They use them to communicate cluster status changes to each other."

"A cluster-receiver channel (CLUSRCVR) channel definition defines the receiving end of a channel on which a cluster queue manager can receive messages from other queue managers in a cluster. A cluster-receiver channel can also carry information about the cluster-information destined for the repository. The definition of a cluster-receiver channel has the effect of advertising that a queue manger is available to receive messages. You need at least one cluster-receiver channel for each cluster queue manager."

If all the repository queue managers go out of service at the same time, queue managers continue to work using the information contained in their partial repositories. New information and requests for updates cannot be processed. When the repository queue managers reconnect to the network, messages are exchanged to bring all repositories (both full and partial) back up to date.

2.4.2 Organizing a cluster

Having selected some queue managers to hold repositories, decide which queue managers should link to which repository. The CLUSSDR channel definition links a queue manager to a repository from which it finds out about the other repositories in the cluster. From then on, the queue manager sends messages to any two ore more repositories, but it always tries to use the one to which it has a CLUSSDR channel definition first. It is not significant which repository is chosen.

It is not advisable to use a repository queue manager on an OS/390 system as the repository queue manager because MQSeries for OS/390 does not have a command server. To ensure that a particular repository queue manager is not used by the MQSeries Explorer, include the string '%NOREPOS%' in the description field of its cluster-receiver channel definition. When the explorer is choosing which repository to link to, it ignores those channel description containing '%NOREPOS%', and treats them as though the queue manager did not hold a repository for the cluster. If there are a large number of repositories or they are spread over a large area, it would be advisable to make a second CLUSSDR channel definition.

Choosing names

When setting up a new cluster, consider a naming convention for the queue managers. Every queue manager must have a different name, but it may help to remember which queue managers are grouped where if given a set of similar names. The queue naming convention of a cluster queue manager follows the same naming convention of any other queue manager. Please refer to section 2.1.2 for queue manager naming conventions. It is recommended that the cluster name be descriptive of the function the cluster is performing. The cluster name is limited in length to 48 characters. For example, the name given to the MQSeries cluster for SFA was "EAI".

- Every cluster-receiver channel must have a unique name. One possibility is to use the queue-manager name preceded by the preposition 'TO'. The name would be of the form:

  FIRSTNODE.SECONDNODE.

  Where:
  - FIRSTNODE is replace with the literal TO.
  - SECONDNODE is replaced with the queue manager name.

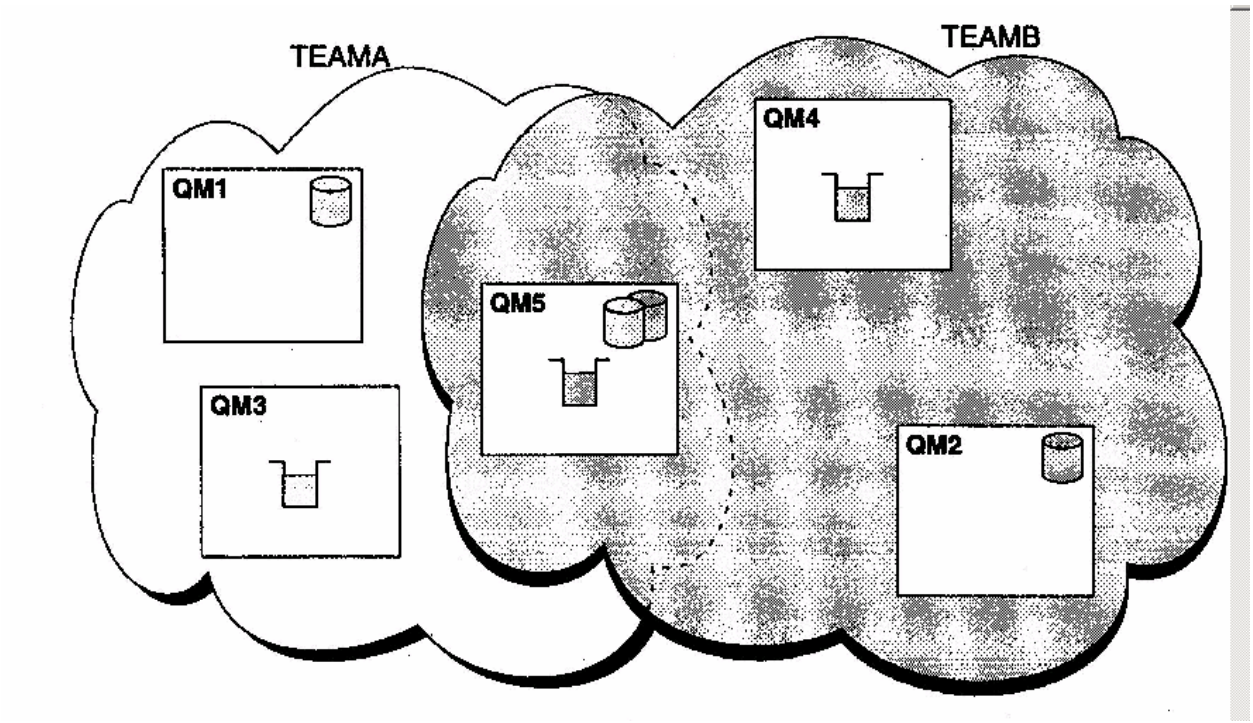
  Example:
  TO.SU35E16
  TO.SU35E17

  Remember that all cluster-sender channels have the same name as their corresponding cluster-receiver channel.
- Do not use generic connection names on cluster-receiver definitions. If a CLUSRCVR is defined with a generic CONNAME there is no guarantee that the CLUSSDR channels will point to the queue managers intended. The initial CLUSSDR may end up pointing to any queue manager in the queue-sharing group, not necessarily one that hosts a repository. Furthermore, if a channel goes to retry status, it may reconnect to a different queue manager with the same generic name and the flow of messages will be disrupted. Basically, the CONNAME should be the network address of the machine the queue manager resides on.

2.4.3 Overlapping clusters

Create clusters that overlap. There are a number of reasons to do this, for example:

- To allow different organizations to have their own administration.
- To allow independent applications to be administered separately.
- To create classes of service.
- To create test and production environments.



In the figure above, the queue manager QM5 is a member of both the clusters illustrated.

If there is more than one cluster in the network, it is essential to give them different names. If two clusters with the same name are ever merged, it will not be possible to separate them again.
When defining a cluster, the following objects are included in the set of default objects defined when creating a queue manager on V5.X of Sun Solaris and Windows NT, and in the customization samples for MQSeries for OS/390.

Do **not** alter the default queue definitions. This could alter the default channel definitions in the same way as any other channel definition, using MQSC or PCF commands.

2.4.4 In the Unlikely Event of a Repository Failure

Cluster information is carried to repositories (whether full or partial) on a local queue called SYSTEM.CLUSTER.COMMAND.QUEUE. If this queue should fill up, perhaps because the queue manager has stopped working, the cluster-information messages are routed to the dead-letter queue. If this is observed from the messages on the queue-manager log or OS/390 system console, an application will need to be executed to retrieve the messages from the dead-letter queue and reroute them to the correct destination.

If errors occur on a repository queue manager, messages will appear defining what error has occurred and how long the queue manager will wait before trying to restart. On MQSeries for OS/390 the SYSTEM.CLUSTER.COMMAND.QUEUE is get-disabled. After identifying and resolving the error, get-enable the SYSTEM.CLUSTER.COMMAND.QUEUE so that the queue manager will be able to restart successfully.

In the unlikely event of a queue manager's repository running out of storage, storage allocation errors will appear on the queue-manager log or OS/390 system console. If this happens, stop and then restart the

queue manager. When the queue manager is restarted, more storage is automatically allocated to hold all the repository information.

2.4.5 Cluster channels

Although using clusters relieves the need to define channels (because MQSeries defines them by default), the same channel technology used in distributed queuing is used for communication between queue managers in a cluster. To understand about cluster channels, become familiar with matters such as:

- How channels operate
- How to find their status
- How to use channel exits

These topics are all discussed in the *MQSeries Intercommunication* book.

When defining cluster-sender channels and cluster-receiver channels, do not set the "disconnect interval" too low (less than about 10 seconds). If it is set too low, the channel may close down between sending a request to a repository queue manager and receiving the response.

If the cluster-sender end of a channel fails and subsequently tries to restart, the restart is rejected if the cluster-receiver end of the channel has remained active. To avoid this problem, arrange for the cluster-receiver channel to be terminated and restarted, when a cluster-sender channel attempts to restart.

On V5.X of MQSeries for Sun Solaris and Windows NT
Control this using the AdoptNewMCA, AdoptNewMCATimeout, and AdoptNewMCACheck attributes in the qm.ini file or the Windows NT Registry. See the *MQSeries System Administration* book for more information.

On MQSeries for OS/390
Control this using the ADOPTMCA and ADOPTCHK parameters of CSQ6CHIP. See the *MQSeries for OS/390 System Setup Guide* for more information.

All documentation referenced above can be found in appendix A

2.5 MQSeries Cluster Implementation Guidelines

- On OS/390 clustering cannot be used if the system is using CICS for distributed queuing. In order to get the most benefit out of using clusters, the queue managers in the network need to be on a platform that supports clusters. Until all the systems are migrated to a platform that supports clusters, the system may have queue managers outside a cluster that are not able to access the cluster queues without extra manual definitions. The clustering facility is available to queue managers on the following platforms:

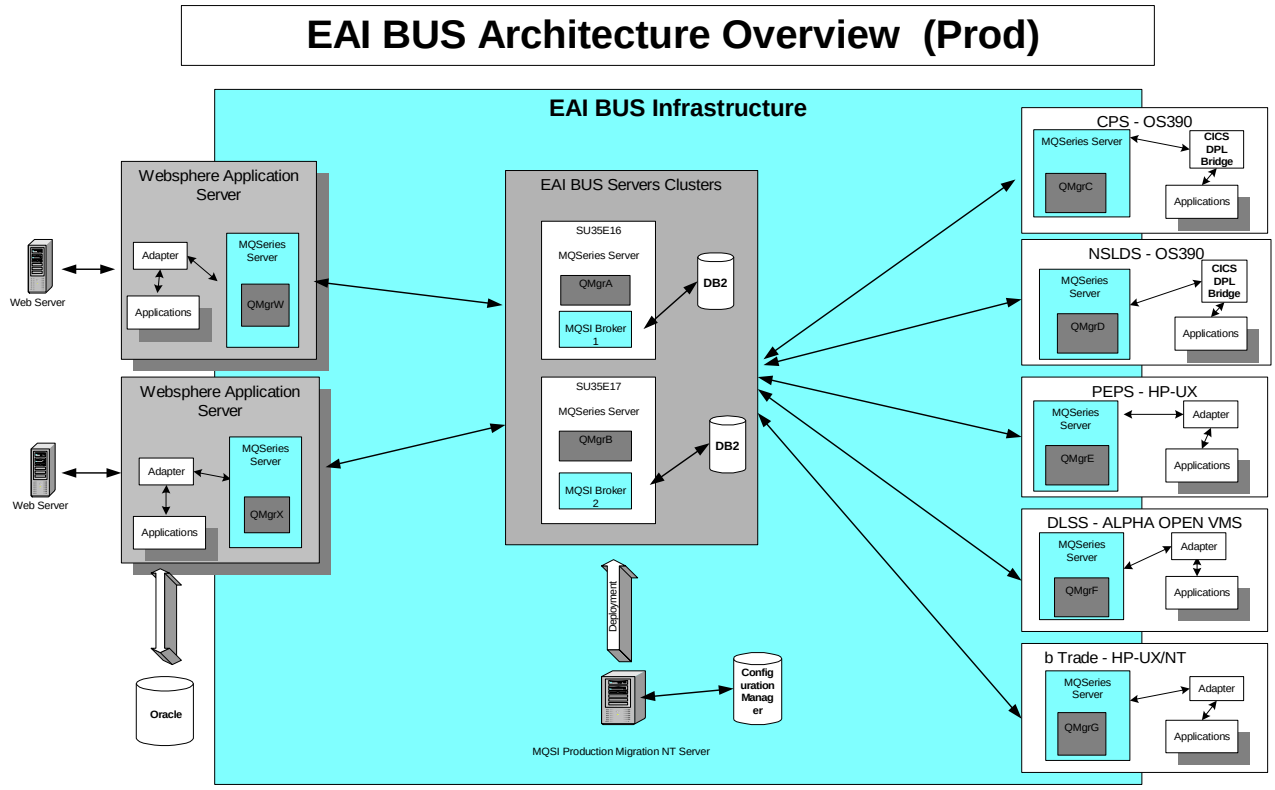
  MQSeries for AIX V5.1
  MQSeries for AS/400 V5.1
  MQSeries for HP-UX V5.1
  MQSeries for OS/2 Warp V5.1
  MQSeries for OS/390
  MQSeries for Sun Solaris V5.1
  MQSeries for Windows NT V5.1

- If two clusters with the same name were merged, it would not be possible to separate them again. Therefore, it is advisable to give all clusters a unique name.
- If a message arrives at a queue manager but there is no queue there to receive it, the message is put to the dead-letter queue as usual. (If there is no dead-letter queue, the channel fails and retries, as described in “Dead-letter queue Guidelines” in the MQSeries Intercommunication book.)
- Using clusters reduces system administration. Clusters make it easy to connect larger networks with many more queue managers than would be possible to contemplate using distributed queuing. However, as with distributed queuing, there is a risk that the system may consume excessive network resources if attempting to enable communication between every queue manager in a cluster.
- The purpose of distribution lists, which are supported on V5.1 of MQSeries for Sun Solaris and Windows NT, is to use a single MQPUT command to send the same message to multiple destinations. Distribution lists can be used in conjunction with queue manager clusters. However, in a clustering environment all the messages are expanded at MQPUT time and so the advantage, in terms of network traffic, is not so great as in a non-clustering environment. The advantage of distribution lists, from the administrator’s point of view, is that the numerous channels and transmission queues do not need to be defined manually.
- If using clusters to achieve workload balancing, first examine the applications to see whether the applications require messages to be processed by a particular queue manager or in a particular sequence. Such applications are said to have message affinities. Applications may need to be modified before being used in complex clusters.
- It is not advisable to use clustering in an environment where IP addresses change on an unpredictable basis such as on machines where Dynamic Host Configuration Protocol (DHCP) is being used.

2.6 SFA Cluster Specifics

2.6.1 Physical layout of the cluster

The hardware architecture implemented at SFA is shown in the diagram below.



2.6.2 Cluster configuration

The SFA EAI cluster, given the name "EAI", includes 3 machines: the WebSphere Application Server, SU35E5 and the two Sun Solaris Servers, SU35E16 and SU35E17. The Sun Servers are the repository queue managers for the cluster.

The steps used in creating the cluster are:

1. Install MQSeries on the system.

2. Create the queue managers and the default objects with the crtmqm command.

3. Start the channel initiator and the channel listener. The channel initiator monitors the system-defined initiation queue SYSTEM.CHANNEL.INITQ which is the initiation queue for all transmission queues. The channel listener must be run on each system. A channel listener program 'listens' for incoming network requests and starts the appropriate receiver channel when it is needed.

4. Decide upon the cluster name, in the case of SFA the name of EAI was chosen for the cluster.

5. Determine which queue managers should hold full repositories. For SFA, both nodes SU35E16 and SU35E17 were chosen to hold full repositories.

6. Alter the queue manager definitions to add repository definitions. The command ALTER QMGR REPOS(EAI) was executed on both SU35E16 and SU35E17.

7. Define the CLUSRCVR channels. For each queue manager in a cluster you need to define a cluster receiver channel on which the queue manager can receive messages. The command was executed on SU35E5, SU35E16, and SU35E17 with the command:
   For example:
   On SU35E5: DEFINE CHANNEL(TO.SU35E5) CHLTYPE(CLUSRCVR) TRPTYPE(TCP) CONNAME(ip address of SU35E5) CLUSTER(EAI)
   On SU35E16: DEFINE CHANNEL(TO.SU35E16) CHLTYPE(CLUSRCVR) TRPTYPE(TCP) CONNAME(ip address of SU35E16) CLUSTER(EAI)
   On SU35E17: DEFINE CHANNEL(TO.SU35E17) CHLTYPE(CLUSRCVR) TRPTYPE(TCP) CONNAME(ip address of SU35E17) CLUSTER(EAI)

8. Define the CLUSSDR channels. On every queue manager in a cluster, you need to define one cluster-sender channel on which the queue manager can send messages to one of the repository queue managers. So,
   On SU35E5: DEFINE CHANNEL(TO.SU35E16) CHLTYPE(CLUSSDR) TRPTYPE(TCP) CONNAME(ip address of SU35E16) CLUSTER (EAI)
   On SU35E16: DEFINE CHANNEL(TO.SU35E17) CHLTYPE(CLUSSDR) TRPTYPE(TCP) CONNAME(ip address of SU35E17) CLUSTER (EAI)
   On SU35E17: DEFINE CHANNEL(TO.SU35E16) CHLTYPE(CLUSSDR) TRPTYPE(TCP) CONNAME(ip address of SU35E16) CLUSTER(EAI)

   Once the queue manager has definitions for both a cluster-receiver channel and a cluster-sender channel in the same cluster, the cluster-sender channel is started.

9. Define any cluster queues. For example:
   On SU35E16: DEFINE QLOCAL(EAI.FROM.WAS.LOAN) CLUSTER(EAI)

2.7 MQSeries Websphere Design Guidelines

The Web Application Server communicates with the EAI bus to retrieve and put information to different legacy data sources. WebSphere Application Server is the standard Java Application Server in the Integrated Technical Architecture (ITA) at SFA. The WebSphere Server will host Web Based applications that act as middleware between the client browser and SFA's Legacy Systems via the EAI bus. Using Java Server Pages, Servlets and Enterprise Java Beans, WebSphere implements SFA's business application logic through Java based Applications. Several methods exist to enable communication between a WebSphere hosted application and the EAI bus.

2.7.1 WebSphere Connectors

2.7.1.1 Common Connector Framework

The Common Connector Framework is a standard for developing applications using E-Business Patterns. When a Web Application Server needs to access a Backend Enterprise Information System, whether it is a middleware messaging system, Enterprise Database system, or a System 390 Transaction Management System, several common communication procedures must take place. These procedures may include starting a transaction, processing data, passing status, and closing the transaction. Whether the backend system is CICS, IMS, DB2 or an Oracle RDBMS, the actual commands and parameters may be different but the high level procedures are common. Since these procedures and backend systems have already

been identified, prebuilt java classes can be written to communicate with these systems. This requires a change to the parameters and data that is passed to the backend systems.

The Common Connector Framework (CCF) is actually implemented within IBM's java development tool, Visual Age for Java (VAJ). The needed classes that implement the binding between the Web Application Server and MQSeries are included within VAJ's Enterprise Access Builder, which is part of VAJ Enterprise Edition. Programs written using the MQSeries CCF connector classes can communicate with MQSeries Applications using the standard MQSeries Programming Interface or the MQSeries Client classes for java interface. A programmer can use the SmartGuide Wizard within VAJ to build a program shell that will communicate with MQSeries and all that is required is to add the application business logic that will make decisions.

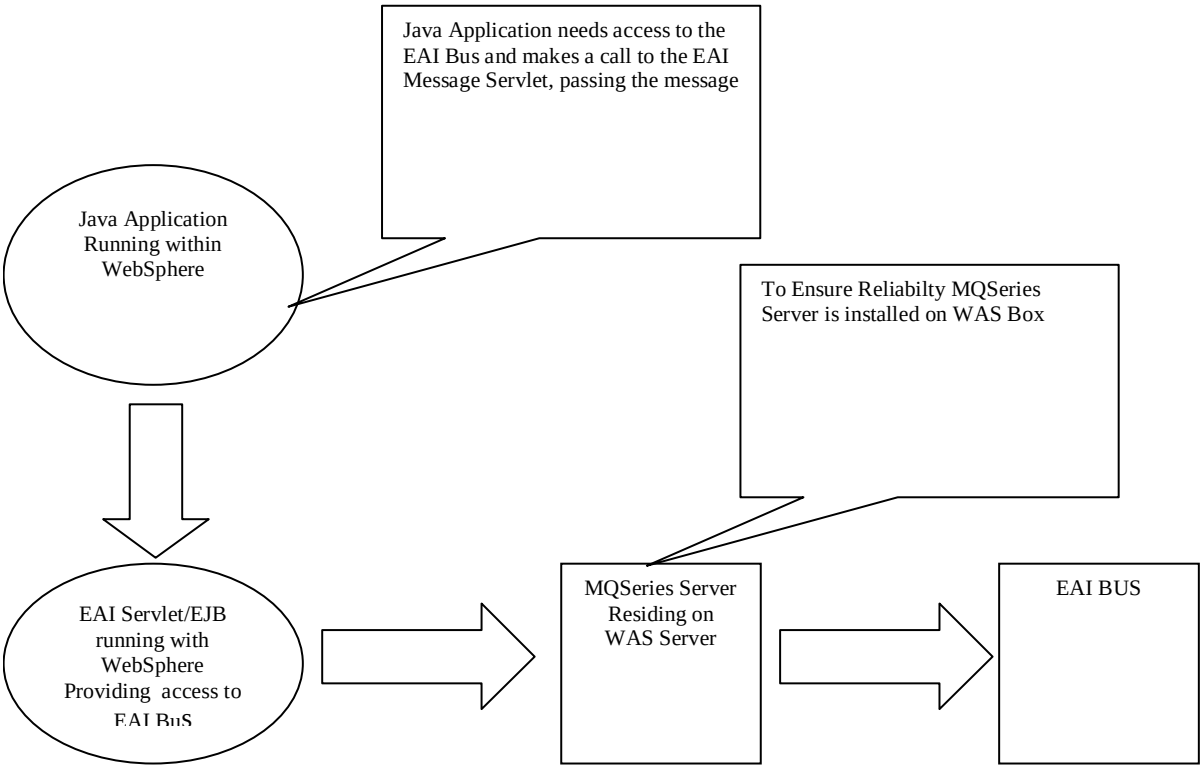
2.7.1.2 Build Your Own Connector

Using MQSeries client classes for Java, a programmer can develop their own interface to MQSeries. This option should only be used by very experienced programmers that have previously implemented Java interfaces to messaging systems. This option is not recommended because the CCF framework is so readily available.

2.7.2 Architecture look and feel

Whether using the CCF framework or building a custom connector, it is important to have a standard, reusable application component within the Java Application Server that enforces communication and data transfer standards between the Application Server and EAI Bus. This reusable component can exist as a Servlet or an Enterprise Java Bean on the WebSphere Application Server. When other applications require access to the EAI Bus, the applications would make a call to the servlet/EJB, which then forwards the message to the EAI bus. This reusable servlet/EJB enforces naming standards, queue names and cluster names before sending message data to the EAI bus. This servlet/EJB would also control the number of connections to MQSeries and allow a central place to tune and manage the web application interface to MQSeries.

To provide reliability and availability of the EAI Bus, the MQSeries Server component should be installed on all WebSphere Application Server (WAS). If an active MQSeries Server with defined Queue Managers are installed on the WAS Server, this ensures assured delivery of all messages to the target destination. If the Queue Manager on the target destination server goes down the sending MQSeries Server will retain the message data and send once connectivity to the target Queue Manager is restored.



2.7.3 SFA EAI WebSphere Reusable Component

The Release One EAI Core Architecture team has developed a reusable WebSphere Java component as an aid to SFA application developers in connecting Internet applications to the EAI Bus. This reusable component is written in Java. It provides a class file for putting and getting messages into and out of a MQSeries message queue. The application incorporates this Java class within the application code to provide a transparent mechanism for putting messages into a queue to be sent to the EAI Bus for processing, and to retrieve message data from a queue upon return. The application specific logic must be built into the application to pass the required message data and to process the message data upon receipt.

2.7.3.1 WebSphere MQ Adapter Overview

The WebSphere MQ Adapter is a Java component that provides a Class file to put message data into a MQSeries message queue and to get message data from a MQSeries message queue. The adapter utilizes MQSeries MQI calls to perform this functionality. In addition, the MQ Adapter provides XML translation capability to transform the input message data from the WebSphere server application into the application specific XML format required to validate the EAI Core architecture for Release One. The input data can be of any format, as long as the XML mapping is defined in the application specific MQ Adapter. Then the message data can be passed to the EAI Bus for transformation by MQSI. This functionality was provided for the PEPS and bTrade validation test.

The developed EAI MQ Adapter resides in the ClearCase repository. Any SFA application development team can utilize this functionality for putting messages into a MQSeries message queue, transforming into XML format, and getting the returned message data from the legacy system for processing by the application.

3. APPLICATION INTEGRATION

This section describes the guidelines to integrate an application into the SFA EAI Bus Architecture. Many aspects into integrating an application into the EAI Bus will depend on the application. Some components may or may not be required and some additional components may be required depending on the application. There are five legacy systems which have the basic infrastructure defined to hook into the EAI Bus. Although the infrastructure may be defined, it may require modifications depending on varying factors, which include, but not limited to, capacity planning and performance. The following are these five legacy systems:

1. CPS
2. NSLDS
3. DLSS
4. PEPS
5. bTrade

3.1 Legacy System Application EAI Integration Considerations

This section will provide overall guidelines for beginning the process of integrating applications at SFA and utilizing the EAI Bus, as developed for the Release One EAI Core Architecture. To integrate an application on any of the Release One legacy system, first determine the application EAI requirements. Does it require an existing adapter, developed as part of the EAI Core architecture, or some modifications to use MQSeries API calls? What is the volume of data travelling through the bus? What are the message sizes? Frequency? What are the application security requirements? Determine the message formats – input and output.

The following is a starting list of EAI Bus components to review when integrating the application:

1. What are the applications requirements
2. Are additional MQSeries objects required
3. Can the application utilize the core adapter

Once the application requirements are clearly defined as they pertain to the EAI, the application team should begin their EAI application design and development phase. For the Release One of the EAI Core Architecture MQ Adapters were built for each release one legacy system to validate the MQ Messaging infrastructure and the functionality to execute a sample non-application specific function on each legacy system. These developed MQ Adapters provide a starting point to be used by the application development teams, as a baseline to determine their applicability or the amount of modifications required reusing these adapters for each application.

The following sections provide an overview of what was developed for each Release One legacy system and an overview of each of the developed MQ Adapters.

3.2 CPS

The CPS system has the MQSeries CICS Bridge and DPL adapter installed. Applications that can utilize the DPL adapter cannot use any CICS terminal related commands and must be able to communicate via the CICS COMMAREA. To help guide the design considerations of an application to use the CICS DPL Bridge the "MQSeries Application Programming Guide" and "MQSeries Application Programming Reference" books can be used. References to all IBM manuals can be found in Appendix A.

For an application to use the CPS EAI Bus connectivity, the requirements of the application must be defined.

Depending on the application requirements, it may or may not require additional MQSeries queue objects be defined. If the application is to communicate with another system, then for each system a local queue and remote queue must be defined. At least one for input messages and one for output messages. If an additional input queue object is required, it will require an additional CICS DPL Bridge Monitor transaction defined and configured to automatically start. Each CICS DPL Bridge Monitor transaction will monitor one queue object.

An application that can be supported as a DPL application by the CICS DPL Bridge is required to use the CICS COMMAREA to communicate. It cannot contain any CICS terminal related commands. Although, to hook it into the EAI Bus, format the request message with the required message structures. It is recommended to build these structures into the MQSI message flow, since it supports these structures with little effort. The main structure that can be used with the CICS DPL Bridge is the MQSeries CIH header. The use of a header is optional and was not used at SFA, however, the application may require its use. The CICS bridge application must set a number of fields in the MQMD and the MQCIH in order to use the bridge successfully. For more information, see the MQSeries Application Programming Guide.

If the application can utilize the CICS DPL Bridge, all the work will be required from the system that sends the request into CPS. The MQSI message flows would be used to build the common structures, such as the MQSeries CIH structure.

The MQ Adapter developed for the CPS system utilizes the MQ CICS DPL Bridge. A request message, consisting of a SSN and a name id, is originated from the source application. The message is routed through the MQSeries messaging infrastructure using the EAI bus infrastructure. The message is transformed by MQSI to append the required records to the message data, as expected by the CICS program on the CPS system. The message is then routed from the EAI Bus to the CPS system using the MQSeries CICS DPL Bridge. The CICS DPL Bridge enables the originating application to access information residing in the CPS OS/390 platform by invoking the CICS program running in the CICS environment to process the message and send a reply back.

In order to integrate applications on the CPS system through the EAI Bus the application team must first identify the required CICS transactions, if they exist, and configure the CICS DPL Bridge. The input messages will be retrieved and the message ID will be evaluated to call the appropriate CICS program. For those cases where an existing CICS program does not exist, the application development team will have to develop a CICS application to provide the required functionality. Either case requires the configuration of the CICS DPL Bridge to map to the appropriate CICS program.

Input and output for using the CPS MQ Adapter:

Inputs: Message data from the queue. This data is application dependent. Maximum length of application data can be up to 32k.

Outputs: Application specific data is written to a MQSeries queue.

3.3 NSLDS

The NSLDS system has the MQSeries OS/390 Batch Trigger Monitor, two custom batch adapters and a Cool:Gen adapter installed. Applications that can utilize these adapters use some sort of FTP process or CICS Cool:Gen transaction today. For an application to use the NSLDS EAI Bus connectivity, the requirements of the application must be defined.

Depending on the application requirements, it may or may not require additional MQSeries queue objects defined. If the application is to communicate with another system, then for each system a local queue and remote queue must be defined. The requirements may need to have additional MQSeries OS/390 Batch Trigger Monitors set-up. The MQSeries OS/390 Batch Trigger Monitor will monitor one queue object. The Cool:Gen adapters will need to be generated from the Cool:Gen product set.

An application that can use the custom batch adapters will expect a flat file as input and will produce a flat file as output. The applications input file will be created from one of the adapters. The adapter will pull the input file from MQSeries as messages and create the actual application input file. This will allow the application to process the file as input. The application would create an output file to be transported as the reply back through the Bus. The creation of the application input file causes the application to start using the NSLDS CA7 product. The second adapter will be executed as the last step of the applications job. The second adapter will pull the file and push it through MQSeries as messages.

For NSLDS transactions an application that can use the Cool:Gen adapter will require a Cool:Gen CICS transaction be prsent on the NSLDS system. This is the Cool:Gen CICS transaction that has been generated to use the EAI Bus with MQSeries. The Cool:Gen MQSeries adapters are generated from the Cool:Gen toolset for each of the Cool:Gen CICS transactions that are required to be accessible from the EAI Bus.

For NSLDS batch processing any application that currently uses an FTP type process, can utilize these custom adapters to replace the FTP logic.

3.3.1 NSLDS Batch

For the NSLDS batch processing, two MQ Adapters and an OS/390 trigger monitor were developed. The two adapters are NSBATCH1 and NSBATCH2, both Cobol programs. The NSBATCH1 adapter reads messages from a MQSeries queue and writes them to a file. The NSBATCH2 adapter reads data from a file and puts the data as a message to a MQSeries queue.

NSBATCH1: This program reads messages off an inbound queue, parses the messages to extract PELL records and writes the PELL records to a flat file.

ARB62000: Executes Procedure: ARB62000 This is an existing multi-step process that does edit checks against the PELL data and creates a flat file of exception/error records for those Pell records that fail the edits.

NSBATCH2: This program reads in the exception/error file created in ARB62000 and builds outbound messages stringing multiple exception/error records and puts the messages to the outbound reply queue.

OS390 Batch Trigger Monitor

The Trigger Monitor allows a batch application process to be submitted automatically when a message arrives on an inbound application queue.

When the Pell message arrives from the test application the Queue manager, NTT1, puts the message to the inbound applications queue, NSLDS.FROM.EAI.REQPELL.

Since the inbound queue is defined for triggering ‘first’ the Queue Manager will put the data defined in the Process definition: NSLDS.PELL.PROCESS as a message to the intiation queue, NSLDS.BATCH.INIT, when the first message arrives in the queue.

The Batch Trigger Monitor monitors the initiation queue and when a message arrives the Batch Trigger monitor executes the batch process and supplies the Qmgr name and queue name to the adapter, NSBATCH1.

NSBATCH1

One control message is sent from the source server with each PELL file. The control message contains information to tell the program NSBATCH1 how to process the PELL records off the inbound queue.

CM-MSG-CNT – tells how many data messages to process.

CM-MAXMSG-LEN – tells when to stop parsing PELL records from a single message.

CM-TOTAL-REC-CNT– also tells when to stop parsing PELL records from all messages.

CM-LRECL – tells the length of the records to be parsed.

NSBATCH1 does an initial keyed get of msgid = 1 to retrieve the control message as the first message. Using the values from the control message, it pulls the messages from the inbound queue until msg-count = CM-MSG-CNT. For each message, it parses the PELL records from the message for a length of CM-LRECL and writes the records to the PELL output file until CM-TOTAL-REC-CNT or CM-MAXMSG-LEN is reached.

NSBATCH1 writes out the control message to a control file to be used by the NSBATCH2 program.

Inputs: The program expects 2 messages as input:

1) Control message:

   Record length: 177 bytes character format

   Purpose of input parameter is to tell the adapter:

   1. Number of data messages
   2. Max data message length
   3. Number of records in data message
   4. Length of data records

2) Data message
   Pell Grant request records
   Record length: 300 bytes character format

Outputs: A flat file is generated as output. It's a file of Pell request records.

NSBATCH2

The control file created in NSBATCH1 contains information to tell the program NSBATCH2 how to build the messages for the put to the outbound queue.

CM-MAXMSG-LEN – tells when to stop reading error records and stringing them out an outbound message.

CM-LRECL – tells the length of the records to be strung together

NSBATCH2 reads the control file once (only one record) and moves values to WS. The program then reads the error file and moves each record out to the outbound message buffer by the length of CM-LRECL until CM-MAXMSG-LEN or end-of-file condition on the Error file. Then put the messages to the outbound replyto queue.

To compile NSBATCH1 and NSBATCH2: MQADM2.A.JCLLIB(CMPJCL02) specify which program needs be compiled.

Inputs: The program expects one file as input.

File description: Errors generated from processing Pell requests through the existing NSLDS batch job: ARB62000.

The filename is limited to a maximum length of 109 characters.

The purpose of input file is to send back Pell requests that are in error back to the test application.

Outputs: Message is written to a MQSeries queue.

3.3.2 NSLDS Transaction

The MQ Adapter developed for the NSLDS Transaction capability was built using the Cool:Gen development tool set. For any Cool:Gen transactions to be generated to use the EAI Bus the application development team must have the Cool:Gen development tools and the capability to execute Cool:Gen applications. The Cool:Gen application is deployed onto the source server as well as the NSLDS mainframe system. The source server generates the transaction data, puts the message data into a MQSeries message queue, using the generated Cool:Gen MQ Adapter, routes the message to the NSLDS system for processing, and retrieves the message data from the message queue via Cool:Gen MQ Adapter deployed onto the mainframe system. The Cool:Gen server executes the specified Cool:Gen CICS transaction and returns the results back to the source server for processing/display.

3.4 DLSS

The DLSS system has a MQSeries trigger monitor and two custom adapters installed. Applications that can utilize these adapters use some sort of file transfer process today. For an application to use the DLSS EAI Bus connectivity, the requirements of the application must be defined. If the application is to communicate with another system, then for each system a local queue and remote queue must be defined.

The DLSS MQ Adapter receives a message containing a SSN and transforms this message, via MQSI into the expected format for the DLSS application. The DLSS application is a COBOL program and has a very extensive record structure as input. An input message consisting of the SSN is sent from the source system to the EAI bus. In the MQSI message flow, the message is transformed by appending required records to the message. The message is then be sent to the DLSS system where a MQ adapter will read from the queue and output the data to a flat file and then release the sequence of programs from the OpenVMS batch queue. The DLSS application processes the file and in turn creates another file with the detail corresponding to each SSN initially sent to the DLSS system. Following the execution of the application, the adapter is run to read from the file created by the DLSS application and puts the message(s) on the queue to be sent back to the source system. Once the batch jobs have been released, there is a five-minute waiting period for an output file to be generated. If no output file is found, an error message is returned to the source system.

Inputs: The program expects one (1) parameter as input and one (1) filename to read data from.

(1) Filename to read data from

Type of input parameter char[500] – character array of size 500.

Maximum length of filename is 500 characters

Purpose of input parameter: Tells the adapter the filespec to read data from.

Outputs: Message is written to a MQSeries queue.

In addition to the MQ Adapters described above, the EAI Core team has developed another set of MQ adapters to execute a real-time DLSS transaction. In the case of the real-time transaction, a message is sent to the DLSS system and the message data is read from the queue by the adapter. Then, a DLSS transaction/job is executed, results are put back on a message queue, and returned to the source system for processing/display.

3.5 PEPS

The PEPS system has a MQSeries trigger monitor and a custom adapter installed. Applications that can utilize this adapter will need to be stored procedures. For an application to use the PEPS EAI Bus connectivity, the requirements of the application must be defined.

Depending on the application requirements, it may or may not require additional MQSeries queue objects defined. If the application is to communicate with another system, then for each system a local queue and remote queue must be defined. One for input messages and one for output messages. The requirements may need to have additional MQSeries Trigger Monitors set-up. The MQSeries HP-UX Trigger Monitor will monitor one queue object. If the application is to deliver messages to another system, then another remote queue will need to be defined.

The MQ Adapter developed as part of the Release One EAI Core Architecture provides a custom developed Java adapter which receives an input message, and executes an Oracle stored procedure to process the request. The PEPS MQ Adapter is a reusable component for any SFA applications that require access to the PEPS database via a stored procedure. The adapter calls the stored procedure to extract the requested database information and returns the results back to the source application. The stored procedure developed for the EAI Core validation is not a production feature so any future application development would require any analysis of the data access requirements from the PEPS system and the development of the required stored procedures. The input data to the PEPS system is in XML format.

3.6 bTrade

The bTrade system has a MQSeries trigger monitor and a custom adapter installed. Applications that can utilize the use of the adapter will be able to pull bTrade data from a bTrade mailbox. For an application to use the bTrade EAI Bus connectivity, the requirements of the application must be defined.

Depending on the application requirements, it may or may not require the definition of additional MQSeries queue objects. If the application is to communicate with another system, then for each system a local queue and remote queue must be defined. One for input messages and one for output messages. The requirements may need to have additional MQSeries Trigger Monitors set-up. The MQSeries HP-UX Trigger Monitor will monitor one queue object.

bTrade.com has provided a Java based application connectorAPI to support the retrieval of messages from a bTrade.com mailbox. The EAI Core Architecture team has developed a MQ Adapter to interface with the bTrade application, through the bTrade connectorAPI, to extract data from a mailbox on the bTrade server.

The bTrade MQ Adapter receives a message request data, in XML format, to retrieve data from a mailbox on the bTrade server. The input message specifies the name of the mailbox to retrieve the data from on the bTrade server. The MQ Adapter developed for the bTrade server will receive the message, route the message data through the EAI Bus for transformation via MQSI, call the bTrade connectorAPI with the appropriate parameters. The connectorAPI will check the status of the specified mailbox, perform the

appropriate action, and return results back to the MQ Adapter. The results will be put into a message queue, sent back to the calling application for processing.

Errors will be reported in the MQRbTrade debug output (tr1debug.out and tr2debug.out) and the MQRbTrade XML replies (tr1xml.out and tr2xml.out) on the bTrade server.

All application and MQ errors and exceptions are printed to the MQbTrade standard error and standard output streams. Additionally, all application and MQ errors and exceptions are returned in the MQ reply-message XML status element. The exception is any error that makes it impossible to either return a MQ reply message or successfully issue a connectorAPI endMessage for retrieval of a single bTrade.com mailbox message.

Inputs to the MQ Adapter on the bTrade server are XML data with the required mailbox name and parameters as specified in the bTrade specification document.

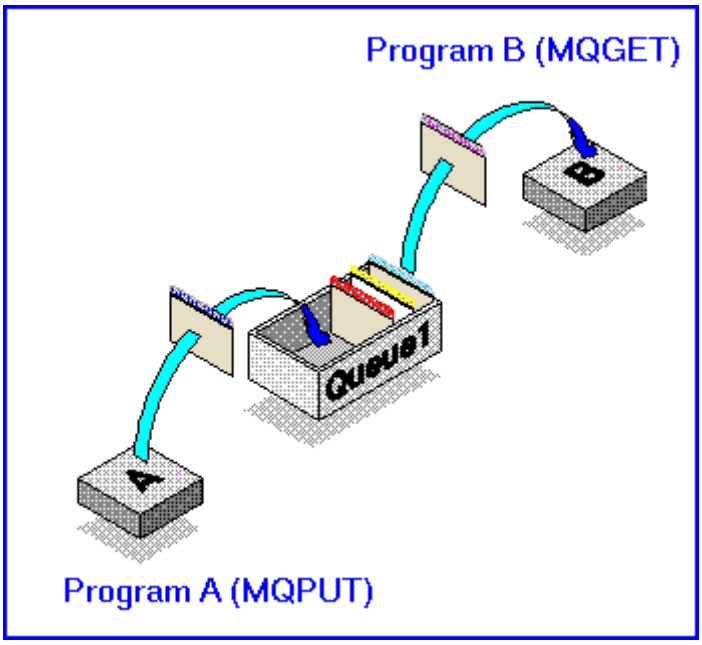
4. SFA APPLICATION ENABLEMENT GUIDELINES

4.1 Application Programs and Messaging

The IBM MQSeries range of products provides application-programming services that enable application programs to communicate with each other using messages and queues. This form of communication is referred to as commercial messaging. It provides assured, once-only delivery of messages. Using MQSeries means that you can separate application programs, so that the program sending a message can continue processing without having to wait for a reply from the receiver. If the receiver, or the communication channel to it, is temporarily unavailable, the message can be forwarded later. MQSeries also provides mechanisms for providing acknowledgements of messages received.

The programs that comprise a MQSeries application can be running on different computers, on different operating systems, and at different locations. The applications are written using a common programming interface known as the Message Queue Interface (MQI), so that applications developed on one platform can be transferred to another.

This figure shows that when two applications communicate using messages and queues, one application puts a message on a queue, and the other application gets that message from the queue.



4.2 Application Usage Guidelines

A queue is a MQSeries object owned by a queue manager, upon which applications can put or retrieve messages. Applications access a queue by using the Message Queue Interface (MQI). Before a message can be put on a queue, the queue must already exist. Each queue must have a name that is unique to the owning queue manager. Before an application can use a queue, it must open the queue, specifying what it wants to do with it. For example, the application can open a queue to:

- Browse messages only (do not delete them)
- Retrieve messages

- Put messages on the queue
- Inquire about the attributes of the queue
- Set the attributes of the queue

For a complete list of the options related to opening a queue, see the description of the MQOPEN call in the *MQSeries Application Programming Reference* manual.

There are different types of queues. These types include:

- *Local*: a local queue is managed by the queue manager to which the application is connected
- *Remote*: a remote queue is managed by a queue manager other than the one to which the application is connected
- *Alias*: an alias queue points to another queue
- *Model*: a model queue is a template for queue definition
- *Dynamic*: a dynamic queue is a temporary queue defined based on a model queue

In SFA's technical environment, the use of alias queues is discouraged, unless a business need dictates its use (e.g. limiting security access to certain queues). Applications putting messages to remote queues will use the remote queue definition. This allows the application to only specify the remote queue name and not be required to know the remote queue manager name. Model and dynamic queues should be used only when a business need dictates their use.

4.2.1 Identifying an Application for a Queue Manager

Any MQSeries application must make a successful connection to a queue manager before it can make any other MQI calls. When the application successfully makes the connection, the queue manager returns a connection handle. This is an identifier that the application must specify each time it issues a MQI call. An application can connect to only one queue manager at a time* (known as its local queue manager), so only one connection handle is valid (for that particular application) at a time. When the application has connected to a queue manager, that queue manager processes all the MQI calls that the application issues until the application issues another MQI call to disconnect from that queue manager. Each adapter written for SFA performs the task of connecting to the queue manager.

* When an application connects to a queue manager, it issues a MQCONN call. The scope of a MQCONN call is limited to the thread that issued it within all of the following:

- MQSeries for AS/400
- MQSeries for Compaq (Digital) OpenVMS
- MQSeries for OS/2 Warp
- MQSeries on UNIX systems
- MQSeries for Windows
- MQSeries for Windows NT

That is, the connection handle returned from a MQCONN call is valid only within the thread that issued the call. Only one call may be made at any one time using the handle. If it is used from a different thread, it will be rejected as invalid. If the application has multiple threads and each wishes to use MQSeries calls, each one must individually issue MQCONN. Each thread can connect to a different queue manager on OS/2 and Windows NT, but not on OS/400 or UNIX. If the application is running as a client, it may connect to more than one queue manager within a thread. This does not apply if the application is not running as a client.

4.2.2 Opening and Closing Queues

Before opening a queue using the MQOPEN call, the application must connect to a queue manager. The application can then use the MQOPEN call to open a queue. The application can also then use the MQCLOSE call to close a queue. When an application opens a queue, the application receives an object handle for that queue. This handle is used in subsequent calls to get or put messages. The same queue can be opened more than once; each open call creates a new object handle. However, most applications will only need to open a given queue once.

Once an application has opened a queue, the application has access to that queue until it closes the queue. The MQOPEN call is costly in terms of time, so once an application has opened a queue and plans to use it in the future, keep the queue open, except when an application only needs to 'put' one message. The MQPUT1 call was designed for this case: this call opens a queue, puts the message, and closes the queue, eliminating the need to use the MQOPEN and MQCLOSE calls.

Queues are automatically closed when an application closes its connection to the queue manager. However, it is a good practice to close all queues before disconnecting from the queue manager.

Each adapter written for SFA performed MQOPEN and MQCLOSE calls.

It is recommended to use the FAIL_IF_QUIESCING open option for the MQOPEN call. This will allow the MQSeries administrators more control of the system.

4.2.2.1 MQOPEN Call

As input to the MQOPEN call, the application must supply:

- A connection handle, using the connection handle returned by the MQCONN call.
- A description of the object to open, using the object descriptor structure (MQOD).
- One or more options that control the action of the call.

The output from MQOPEN is:

- An object-handle that represents access to the queue. Use this as input to any subsequent MQI calls for this queue.
- A modified object-descriptor structure, if the application is creating a dynamic queue.
- A completion code.
- A reason code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed.

4.2.2.2 MQCLOSE Call

As input to the MQCLOSE call, the application must supply:

- A connection handle, using the same connection handle used to open the queue.
- The handle of the queue to close. This comes from the output of the MQOPEN call.

The output from MQCLOSE is:

- A completion code.
- A reason code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed.

4.2.3 Putting Messages On A Queue

To put messages on a queue, an application must use the MQOO_OUTPUT option when issuing the MQOPEN call. After the queue has been opened using this option, the application can issue a MQPUT call to put a message on the open queue. If the application is only putting one message and will not use the queue again, use the MQPUT1 call.

It is recommended to use the FAIL_IF_QUIESCING put-message option for the MQPUT and MQPUT1 calls. This will allow the MQSeries administrators more control of the system.MQPUT call.

As input to the MQPUT call, the application must supply:

- A connection handle, using the connection handle that was returned when the application issued the MQCONN call.
- A queue handle, using the queue handle that was returned when the application issued the MQOPEN call.
- A description of the message the application is putting on the queue. This is in the form of a message descriptor structure.
- Control information, in the form of a put-message options structure. This options structure needs to be redefined for every MQPUT call.
- The length of the application data contained within the message.
- The application data itself.

The output from the MQPUT call is:

- A reason code.
- A completion code.
- If the call completes successfully, it also returns the put-message options structure and the message descriptor structure. One or both structures may have modified attributes within them. For more detail, look at the *MQSeries Application Programming Guide*.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed.

4.2.4 Getting Messages From A Queue

To open a queue so that the messages on that particular queue can be browsed (does not remove the message from the queue), use the MQOPEN call with the MQOO_BROWSE option. To get (and remove) messages from a queue, an application must use the MQOO_INPUT_AS_Q_DEF, MQOO_INPUT_SHARED, or MQOO_INPUT_EXCLUSIVE option when issuing the MQOPEN call. Selection of one of these three options is used to specify if the application opens the queue in exclusive, or shared, mode. See the *MQSeries Application Programming Guide* for more information. After the queue has been opened using one of these options, the application can issue a MQGET call to get a message from the open queue.

By specifying the MsgId and/or CorrelId fields in the message descriptor structure, the application can search the queue for a particular message. If the application uses MQGET call more than once (for example, to step through the messages in the queue), it must set the MsgId and CorrelId fields of this structure to null after each call. This prevents the call from filling these fields with the identifiers of the message that were retrieved, and therefore getting messages with the same identifiers as the previous message.

If the fields in the message descriptor structure are not specified to search for a particular message, the MQGET call will retrieve the first message in the queue.

It is recommended to use the FAIL_IF_QUIESCING get-message option for the MQGET call. This will allow the MQSeries administrators more control of the system.

4.2.4.1 MQGET Call

As input to the MQGET call, the application must supply:

- A connection handle, using the connection handle that was returned when the application issued the MQCONN call.
- A queue handle, using the queue handle that was returned when the application issued the MQOPEN call.
- A description of the message the application wants to get from the queue. This is in the form of a message descriptor structure.
- Control information in the form of a get-message options structure. This control information describes if the application is browsing or removing messages. The control information also describes if the MQI call waits (and how long it waits) for a message or if the call returns immediately.
- The size of the buffer you have assigned to hold the message.
- The address of the storage location in which the message must be put.

The output from the MQGET call is:

- A reason code
- A completion code
- The message in the buffer area specified, if the call completed successfully
- The options structure, modified to show the name of the queue from which the message was retrieved.
- The message descriptor structure, with the contents of the fields modified to describe the message that was retrieved
- The length of the message
- Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed

4.2.5 Queue Manager Connectivity Guidelines

A queue manager supplies applications with MQSeries services. An application must have a connection to a queue manager before it can use the services of that queue manager. An application can make this connection explicitly (using the MQCONN call), or the connection can be made implicitly. For example, CICS for MVS/ESA and CICS/MVS programs do not need to explicitly connect to a queue manager, because the CICS system itself is connected to a queue manager. However, for portability it is recommended that CICS for MVS/ESA and CICS/MVS programs use the MQCONN and MQDISC calls.

4.2.6 Connecting To and Disconnecting From a Queue Manager

To connect to a queue manager, an application must use the MQCONN call. To disconnect from a queue manager, an application must use the MQDISC call.

MQCONN Call

As input to the MQCONN call, the application must supply a queue manager name. To connect to the default queue manager, specify a queue manager name consisting entirely of blanks or starting with a null character.

The output from MQCONN is:

- A connection handle, using this handle in subsequent MQI calls associated with this queue manager.
- A completion code.
- A reason code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed. If the reason code indicates that the application is already connected to that queue manager, the connection handle that is returned is the same as the one that was returned when the application first connected. So the application probably should not issue the MQDISC call in this situation because the calling application will expect to remain connected. The MQCONN call fails if the queue manager is in a queuing state when issuing the call, or if the queue manager is shutting down.

MQDISC Call

As input to the MQDISC call, the application must supply the connection handle that was returned by MQCONN when the application connected to the queue manager.

The output from MQDISC is:

- A completion code.
- A reason code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed.

All adapters written for SFA had to connect to the queue manager, open a queue, perform a MQGET or MQPUT, close a queue, and disconnect from the queue manager. Each future adapter written for SFA will also need to perform each of the above in order to get and put messages on a queue.

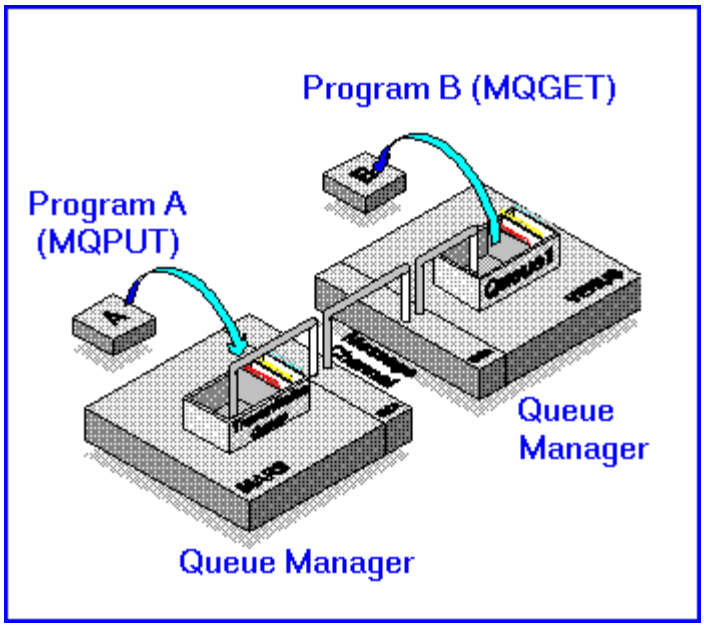
4.2.7 Pass the Connection Name as a Program Parameter

This allows a program to run unchanged on any Queue Manager. This provides the capability for multiple concurrent instances; or a queue driven application could be moved to a different queue manager without impacting the application code.

4.2.8 Messaging Using More Than One Queue Manager

This arrangement is not typical for a real messaging application because both programs are running on the same computer, and connected to the same queue manager. In a commercial application, the putting and getting programs would probably be on different computers, and so connected to different queue managers.

This figure shows how messaging works when the program putting the message and the program getting the message are on the different computers, and are connected to different queue managers.



In this situation, it is necessary to create message channels to carry MQSeries messages between the queue managers.

4.3 Application Interface Programming Options

There is a wide range of options for communicating with MQSeries programs including new interfaces for message content as well message delivery. Programs written using any of these message delivery styles can communicate with each other and with programs written in any of the other MQSeries delivery styles.

4.3.1 Message Delivery

4.3.1.1 MQI

The Message Queue Interface (MQI) is the common API across all platforms. The calls made by the applications running on each platform are common. This allows application programmers to focus on the business logic of the application, rather than the interface differences of each platform. This makes it much easier to write and maintain applications, as well as facilitate migration of applications from one platform to another as required by changing business needs. Each adapter written for SFA made use of a majority of the MQI function calls as shown below. The following figure represents the MQI.

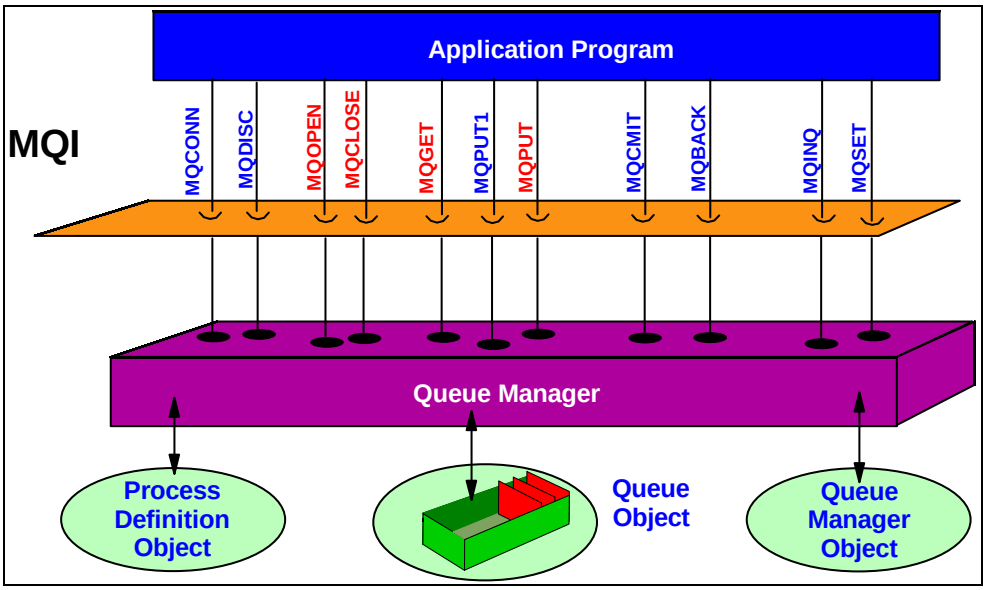


Figure 1 – Message Queue Interface

4.3.1.2 JMS

Java Message Service (JMS) is supported by a MQSeries implementation of this Java standard API for Enterprise Messaging Services. Using JMS, applications can communicate with other MQSeries JMS applications, with applications written to the MQI, or to the Application Message Interface (AMI).

4.3.1.3 AMI

The Application Messaging Interface (AMI) provides a simpler and higher-level programming interface than the MQI. Although it has some limitations compared with the MQI, its function should be sufficient for the majority of users. The AMI supports both point-to-point and publish/subscribe messaging models. The AMI eliminates the need for application programmers to understand all of the options and functions available in the MQI. This was not used at SFA, but is mentioned for future use if the need arises.

The MQSeries AMI can be used to build client applications, and the AMI will automatically build any required headers as specified using the AMI, including the new RFH2 headers. The AMI is designed to simplify the task of the application programmer, while enabling the more advanced functions and message broker facilities to be used.

AMI is a high level API that moves many functions normally performed by messaging applications into the middleware layer, where a set of policies defined by the enterprise is applied on the application's behalf. Policies hold details of how messages are to be handled, for example, priority, confirmation of delivery, timed expiry.

4.3.2 Message Content

4.3.2.1 XML

Extensible Markup Language (XML) is an industry-wide standard for self-defining messages. It enables diverse systems and databases to understand each other's data (for example, to identify fields) by indicating both the content and the role of the data.

XML is supported in MQSeries Integrator Version 2 and MQSeries Workflow Version 3.2; XML will be supported within MQSeries Messaging via the Common Messaging Interface.

For SFA, all messages passed into MQSeries Integrator were in XML. IBM is not advocating the use of XML and the adoption of XML as a standard is outside the scope of this document.

Sample XML Message:

```
<?xml version ="1.0"?>
<!DOCTYPE Message SYSTEM "C:\TestEnvironment\XMLFiles\LifeQuote.dtd">
<!--Generated by XML Authority.-->
<Message issuedTime = "string" Authorisation = "string" sessionID = "string" creationTime = "string"
issueProgram =
"string" issueUser = "string" ID = "id1" issueSystem = "string" txnScope = "string" eventID = "string"
zoneOffset = "string"
language = "string"><!-- (Command.valueQuoteRequest* , Command.valueQuoteResponse* )-->
<Command.valueQuoteRequest responseDTD = "string" echoBack = "string" cmdMode = "always" ID =
"id2"><!--
(%CustomizeAgreement , SystemInfo )-->
<LifeAgreement ID = "id3" REFID = "string" status = "string" UUID = "UUID"><!-- (%Agreement ,
Product
)-->
<policyNumber>only text</policyNumber>
<effectiveFromDate>only text</effectiveFromDate>
<companyCode>only text</companyCode>
<ratingCompany>only text</ratingCompany>
<policyType>only text</policyType>
<renewalDate>only text</renewalDate>
<paymentPlan>only text</paymentPlan>
<agreementState>only text</agreementState>
<lineOfBusinessCode>only text</lineOfBusinessCode>
<effectiveFromDate>only text</effectiveFromDate>
<agentOfRecord>only text</agentOfRecord>
<agentCommission>only text</agentCommission>
<PolicyMessage/>
<MoneyObligation ID="id4" REFID="string" status="string" UUID="UUID"><!--(type, amount,
frequency)-->
<type>only text</type>
<amount>only text</amount>
<frequency>only text</frequency>
</MoneyObligation>
<Discount-Surcharge/>
<Underwriting/>
<Applicant ID = "id5" REFID = "string" status = "string" UUID = "UUID"><!-- (Person )-->
<Person ID = "id6" REFID = "string" status = "string" UUID = "UUID"><!-- (%Party , Body
, PartyActivity* , Residency , PartyContactPointUsage? )-->
<id>only text</id>
<uuid>only text</uuid>
<FamilyName/>
<!-- <UnstructuredName>only text</UnstructuredName> -->
<Body ID = "id7" REFID = "string" status = "string" UUID = "UUID"><!-- (gender ,
height , weight , birthdate , MedicalCondition+ )-->
<gender>Female</gender>
```

```
<height>6.2</height>
<weight>250</weight>
<birthdate>01/01/1980</birthdate>
<MedicalCondition ID = "id8" REFID = "string" status = "string" UUID =
"UUID"><!-- (description , response )-->
<description>High Blood Pressure</description>
<response>Yes</response>
</MedicalCondition>
<MedicalCondition ID = "id9" REFID = "string" status = "string" UUID =
"UUID"><!-- (description , response )-->
<description>Heart Disease</description>
<response>No</response>
</MedicalCondition>

</Body>
```


4.4 EAI Common Error Handling Guidelines

Whenever possible, the queue manager returns any errors as soon as a MQI call is made. The three most common errors that the queue manager can report immediately are described in this section.

4.4.1 Failure of a MQI Call

An example of a MQI call failure is being unable to put a message to a queue because the queue is full. The completion code and return code of the MQI call specify the nature of the failure. Applications should inspect these codes for every MQI call and be able to handle all possible return codes.

4.4.2 System Interruption

The queue manager is an example of a system component needed by the application and when the queue manager is interrupted, the application encounters an error. Applications must ensure no data is lost due to this sort of interruption. To ensure no data loss, applications will get and put messages under syncpoint. This syncpoint activity can be controlled by the queue manager or by some external resource coordinator (e.g. CICS, Encina, etc.).

4.4.3 Unable to Process Messages

Messages containing data that cannot be processed successfully are known as poisoned messages. When applications operate under syncpoint, if the application cannot successfully process a message, the MQGET call is backed out. The queue manager maintains a count (in the BackoutCount field of the message descriptor) of the number of times this happens for MQGET calls which DO NOT use any of the Browse type get message options. Messages whose backout counts increase over time are being repeatedly rejected by the application – the application should be designed to handle such situations. There are many different tactics to handling poisoned messages. One method would be to write the messages to a file and a common “poison message application” attempt to process them at a later point in time. Another method is to have the application itself deal with the message. Messages could also be written to the dead letter queue and then be processed by a dead letter handler. Based on your application requirements a method should be adopted.

4.4.4 Responding to Errors

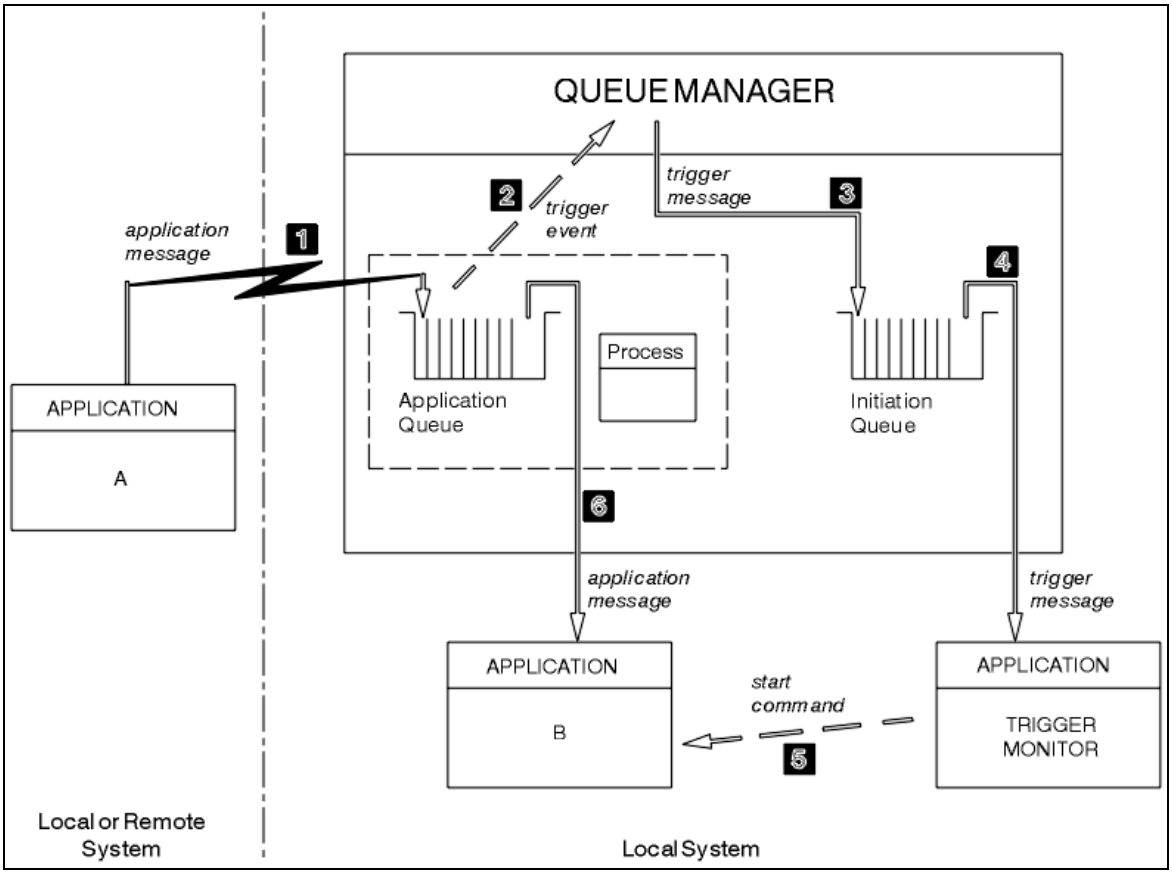
Applications should respond in a similar manner to errors returned by MQI calls. One possible way to implement this common error handling methodology is to provide error-handling routines for the application developer. Use of these common error-handling routines ensures that all application programmers handle MQSeries errors in the same way and do not have to write their own error handling routines.

4.5 Triggered queues and applications

4.5.1 Designing MQSeries Applications

Some MQSeries applications that serve queues run continuously, and are always available to retrieve messages that arrive on the queues. However, this may not be desirable when the number of messages arriving on the queues is unpredictable. In this case, applications could be consuming system resources even when there are no messages to retrieve.

MQSeries provides a facility that enables an application to be started automatically when there are messages available to retrieve. This facility is known as *triggerin*g.



1. Application A, which can be either local or remote to the queue manager, puts a message on the application queue. Note that no application has this queue open for input. However, this fact is relevant only to trigger type FIRST and DEPTH.
2. The queue manager checks to see if the conditions are met under which it has to generate a trigger event. If so, a trigger event is generated. Information that is held within the associated process definition object is used when creating the trigger message.
3. The queue manager creates a trigger message and puts it on the initiation queue associated with this application queue, but only if an application (trigger monitor) has the initiation queue open for input.
4. The trigger monitor retrieves the trigger message from the initiation queue.
5. The trigger monitor issues a command to start application B (the server application).
6. Application B opens the application queue and retrieves the message.

Notes:

1. If the application queue is open for input, by any program, and has triggering set for FIRST or DEPTH, no trigger event will occur since the queue is already being served.
2. If the initiation queue is not open for input, the queue manager will not generate any trigger messages, it will wait until an application opens the initiation queue for input.
3. When using triggering for channels, you are recommended to use trigger type FIRST or DEPTH.

Each adapter created for SFA utilized triggering. The MQSeries object definitions can be seen by viewing each system script file contained in the Clearcase repository. Specifically, you want to look for the objects with the “trigger” attribute.

4.5.2 Starting MQSeries Applications

Trigger messages created because of trigger events that <u>are not</u> part of a unit of work are:

- put on the initiation queue,
- put outside any unit of work, with no dependence on any other messages
- available for retrieval by the trigger monitor immediately

Trigger messages created because of trigger events that <u>are a part</u> of a unit of work are put on the initiation queue, as part of the same unit of work. Trigger monitors cannot retrieve these trigger messages until the unit of work completes. This applies whether the unit of work is committed or backed out. If the queue manager fails to put a trigger message on an initiation queue, it will be put on the dead-letter (undelivered-message) queue.

Notes:

1. The queue manager counts both committed and uncommitted messages when it assesses whether the conditions for a trigger event exist.

   With triggering of type FIRST or DEPTH, trigger messages are made available even if the unit of work is backed out so that a trigger message is always available when the required conditions are met. An example is a put request within a unit of work for a queue that is triggered with trigger type FIRST. This causes the queue manager to create a trigger message. If another put-request occurs from another unit of work, this does not cause another trigger event. Rather, the number of messages on the application queue has now changed from one to two, which does not satisfy the

conditions for a trigger event. If the first unit of work is backed out, but the second is committed, a trigger message is still created.

However, this does mean that trigger messages are sometimes created when the conditions for a trigger event are not satisfied. Applications that use triggering must always be prepared to handle this situation. It is recommended to use the wait option with the MQGET call, setting the *WaitInterval* to a suitable value.

2. For local shared queues (that is, shared queues in a queue-sharing group) the queue manager counts committed messages only.

For SFA, the adapters were triggered on the trigger type of “FIRST”, the queues were then read until empty.

5. APPLICATION CONNECTIVITY (ADAPTERS AND BRIDGES)

The EAI application will be interfacing with several systems. The interfaces between EAI and other systems may require special mechanisms called adapters and bridges.

An adapter or a bridge is a piece of software that moves data between a message on a queue and an application or environment. Adapters handle data inbound-to and outbound-from the application or environment.

5.1 MQSeries Application Adapter

MQSeries provides a mechanism for assured delivery of messages, which can be sent even when the target is disconnected. It can be used to distribute work around a large number of disparate systems in an environment where trying to propagate transactional two-phase commit is not practical.

5.2 Adapter Classifications

5.2.1 Type of Message

Adapters may be classified by the type of message that will be processed:

- **Request/Reply**
  An incoming XML request message from the front-end is posted to the back-end. In response, the adapter always synchronously routes the back-end results in the form of a valid XML document.
- **Fire & Forget**
  An incoming XML request from the front-end is posted to the back-end and no response is required.
- **Notification**
  The adapter routes an incoming message from the back-end to the front-end in the form of a valid XML message. This may be the reply to a message received.

All adapters written for SFA were of the Request/Reply type.

5.2.2 Interface Type

Adapters may be classified by interface type:

- **Java Object -** Creates Java objects that corresponds to the XML message elements.
- **Host structure -**
  1. Converts data from valid XML values to valid host values. Uses tables for simple cases and code for complex transformations.
  2. Creates host objects that correspond to the host data structures and maps the values from the XML objects to the host objects
- **XML Message** – The input data and the output data are both in XML format. The adapter may add the standard header and perform other functions, but does not need to transform the message

5.3 MQSeries-CICS/ESA Bridge

The MQSeries-CICS/ESA Bridge enables an application, not running in a CICS environment, to run a program or transaction on CICS/ESA and get a response back. This non-CICS application can be run from any environment that has access to a MQSeries network that encompasses MQSeries for MVS/ESA.

A program is a CICS program that can be invoked using the EXEC CICS LINK command. It must conform to the DPL subset of the CICS API that is, it must not use CICS terminal or syncpoint facilities.

A transaction is a CICS transaction designed to run on a 3270 terminal. This transaction can use BMS or TC commands. It can be conversational or part of a pseudo conversation. It is permitted to issue syncpoints.

5.3.1 Using the CICS Bridge

Only SFA applications that use a CICS commarea to communicate can utilize the CICS Bridge; any applications that use terminal I/O CICS commands can use the CICS DPL Bridge.

The CICS Bridge allows an application to run a single CICS program or a ‘set’ of CICS programs (often referred to as a unit of work). The adapter written for the CPS system utilizes the CICS Bridge. For more information on the CPS adapter please reference the Technical Specification document. The CICS Bridge works with the application that waits for a response to come back before it runs the next CICS program (synchronous processing). It also works with the application that requests one or more CICS programs to run, but doesn't wait for a response (asynchronous processing).

The CICS Bridge also allows an application to run a 3270-based CICS transaction, without knowledge of the 3270 data stream. The CICS Bridge uses standard CICS and MQSeries security features. It can be configured to authenticate, trust, or ignore the requestor's user ID.

With this flexibility, there are many instances where the CICS Bridge can be used. For example,

- To write a new MQSeries application that needs access to logic or data (or both) that reside on your CICS server.
- Enabling a Lotus Notes application to run CICS programs.
- To be able to access CICS applications from a MQSeries Java client application or a web browser using the MQSeries Internet gateway.

5.3.2 CICS Bridge at Work

This section explains how the CICS Bridge works and the options available when deciding what level of security to use.

With respect to system setup, note the following:

- Ensure that the MQSeries-CICS adapter is enabled.
- The CICS Bridge requires that both MQSeries and CICS are running in the same MVS image.
- The MQSeries request queue must be local to the CICS Bridge, however the response queue can be local or remote.
- The CICS bridge tasks must run in the same CICS as the bridge monitor. The user programs can be in the same or a different CICS system.

5.4 Running CICS DPL programs

Data necessary to run the program is provided in the MQSeries message. The bridge builds a COMMAREA from this data, and runs the program using EXEC CICS LINK.

The following shows the components and data flow to run a CICS DPL program.

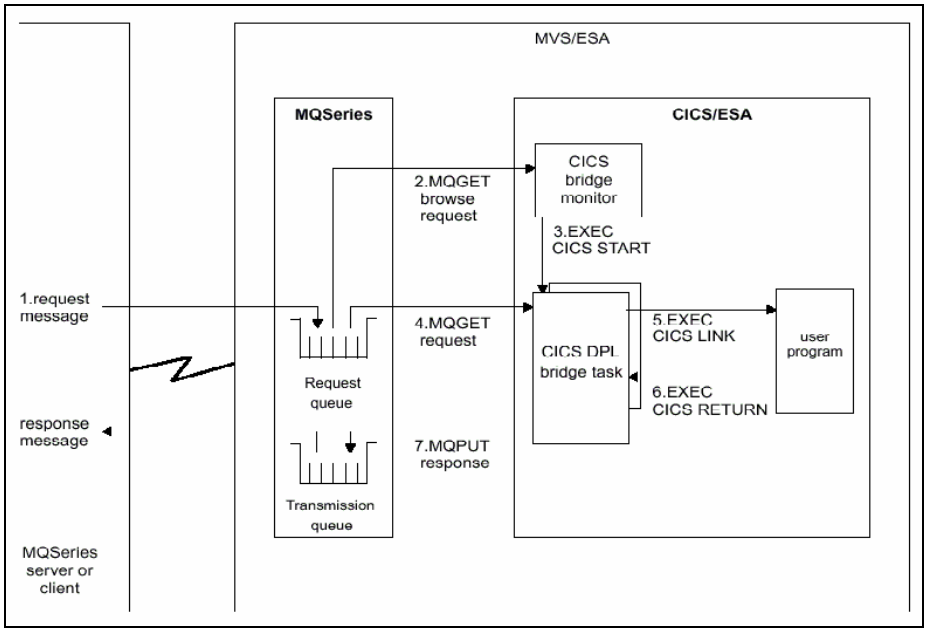


Figure 2 – CICS DPL Transaction

The following takes each step in turn, and explains what takes place:

1. A message, with a request to run a CICS program, is put on the request queue.
2. The CICS Bridge monitor task, which is constantly browsing the queue, recognizes that a 'start unit of work' message is waiting *(CorrelI*d=MQCI_NEW_SESSION).
3. Relevant authentication checks are made, and a CICS DPL Bridge task is started with the appropriate authority.
4. The CICS DPL Bridge task removes the message from the request queue.
5. The CICS DPL Bridge task builds a COMMAREA from the data in the message and issues an EXEC CICS LINK for the program requested in the message.
6. The program returns the response in the COMMAREA used by the request.
7. The CICS DPL Bridge task reads the COMMAREA, creates a message, and puts it on the reply-to queue specified in the request message. All response messages (normal and error, requests and replies) are put to the reply-to queue with default context.
8. The CICS DPL bridge task ends.

A unit of work can be just a single user program, or it can be multiple user programs. There is no limit to the number of messages you can send to make up a unit of work.

5.4.1 Running CICS 3270 transactions

Data necessary to run the transaction is provided in the MQSeries message. The CICS transaction runs as if it has a real 3270 terminal, but instead uses one or more MQSeries messages to communicate between the CICS transaction and the MQSeries application. Unlike traditional 3270 emulators, the bridge does not work by replacing the VTAM flows with MQSeries messages.

Instead, the message consists of a number of parts called vectors, each of which corresponds to an EXEC CICS request. Therefore, the application is talking directly to the CICS transaction, rather than via an emulator, using the actual data used by the transaction (known as application data structures or ADSs).

The following shows the components and data flows to run a CICS 3270 transaction.

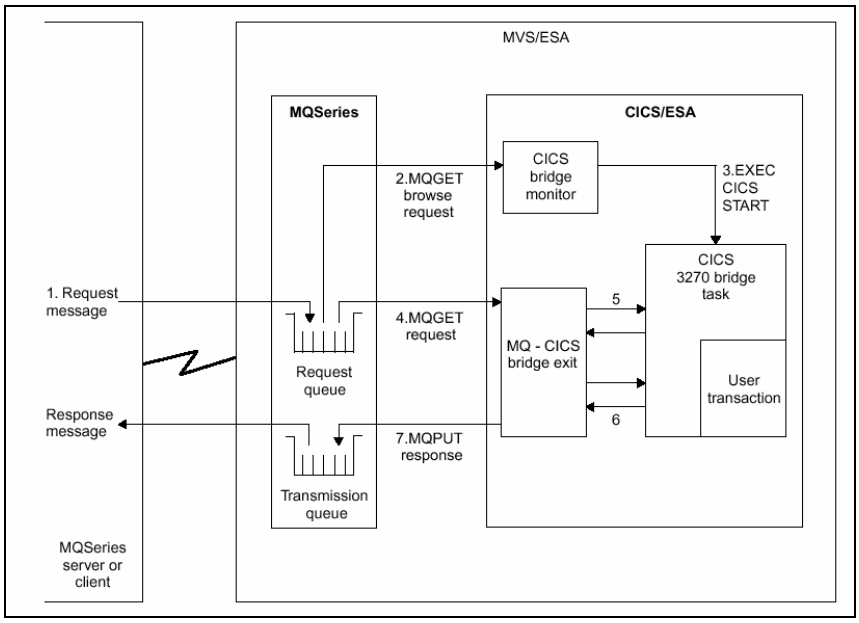


Figure 3 – CICS 3270 Transaction

The following takes each step in turn, and explains what takes place:

1. A message, with a request to run a CICS transaction, is put on the request queue.
2. The CICS Bridge monitor task, which is constantly browsing the queue, recognizes that a 'start unit of work' message is waiting *CorrelI*d=MQCI_NEW_SESSION).
3. Relevant authentication checks are made, and a CICS 3270 bridge task is started with the appropriate authority.
4. The MQ-CICS bridge exit removes the message from the queue and changes task to run a user transaction.
5. Vectors in the message provide data to answer all terminal related input EXEC CICS requests in the transaction.
6. Terminal related output EXEC CICS requests result in output vectors being built.
7. The MQ-CICS bridge exit builds all the output vectors into a single message and puts this on the reply-to queue.

8. The CICS 3270 bridge task ends.

A traditional CICS application usually consists of one or more transactions linked together as a pseudo conversation. In general, the 3270 terminal user entering data onto the screen and pressing an AID key starts each transaction. This model of application can be emulated by a MQSeries application. A message is built for the first transaction, containing information about the transaction, and input vectors. This is put on the queue.

The reply message will consist of the output vectors, the name of the next transaction to be run, and a token that is used to represent the pseudo conversation. The MQSeries application builds a new input message, with the transaction name set to the next transaction and the facility token set to the value returned on the previous message. Vectors for this second transaction are added to the message, and the message put on the queue. This process is continued until the application ends.

An alternative approach to writing CICS applications is the conversational model. In this model, the original message might not contain all the data to run the transaction. If the transaction issues a request that cannot be answered by any of the vectors in the message, a message is put onto the reply-to queue requesting more data. The MQSeries application gets this message and puts a new message back to the queue with a vector to satisfy the request.

6. COMMITTING AND BACKING OUT UNITS OF WORK

This section describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work. The following terms, described below, are used in this section:

- Commit
- Back out
- Syncpoint coordination
- Syncpoint
- Unit of work
- Single-phase commit
- Two-phase commit

6.1 Committing and Backing Out

When a program puts a message on a queue within a unit of work, that message is made visible to other programs only when the program *commits* the unit of work. To commit a unit of work, all updates must be successful to preserve data integrity. If the program detects an error and decides that the put operation should not be made permanent, it can *back out* the unit of work. When a program performs a back out, MQSeries restores the queue by removing the messages that were put on the queue by that unit of work. The way in which the program performs the commit and back out operations depends on the environment in which the program is running.

When a program gets a message from a queue within a unit of work, that message remains on the queue until the program commits the unit of work, but the message is not available to be retrieved by other programs. The message is permanently deleted from the queue when the program commits the unit of work. If the program backs out the unit of work, MQSeries restores the queue by making the messages available to be retrieved by other programs. Changes to queue attributes (either by the MQSET call or by commands) are not affected by the committing or backing out of units of work.

6.2 Syncpoint Coordination, Syncpoint, Unit of Work

Syncpoint coordination is the process by which units of work are either committed or backed out with data integrity. The decision to commit or back out the changes is taken, in the simplest case, at the end of a transaction. However, it can be more useful for an application to synchronize data changes at other logical points within a transaction. These logical points are called *syncpoints* (or *synchronization point*s) and the period of processing a set of updates between two syncpoints is called a *unit of wor*k. Several MQGET calls and MQPUT calls can be part of a single unit of work. The maximum number of messages within a unit of work can be controlled by the DEFINE MAXSMSGS command on OS/390, or by the MAXUMSGS attribute of the ALTER QMGR command on other platforms. See the *MQSeries Command Reference* book for details of these commands.

6.3 Single-phase Commit

A *single-phase commit* process is one in which a program can commit updates to a queue without coordinating its changes with other resource managers.

6.4 Two-phase Commit

A *two-phase commit* process is one in which updates that a program has made to MQSeries queues can be coordinated with updates to other resources (for example, databases under the control of DB2). Under

such a process, updates to *all* resources are committed or backed out together. To help handle units of work, MQSeries provides the *BackoutCount* attribute. This is incremented each time a message, within a unit of work, is backed out. If the message repeatedly causes the unit of work to abend, the value of the *BackoutCount* finally exceeds that of the *BackoutThreshold*. This value is set when the queue is defined. In this situation, the application can choose to remove the message from the unit of work and put it onto another queue, as defined in *BackoutRequeueQName* . When the message is moved, the unit of work can commit.

Transaction managers (such as CICS, IMS, Encina, and Tuxedo) can participate in two-phase commit, coordinated with other recoverable resources. This means that the queuing functions provided by MQSeries can be brought within the scope of a unit of work, managed by the transaction manager.

7. SYNCPOINT GUIDELINES

A MQSeries application can specify on every put and get call whether the call is to be under syncpoint control. To make a put operation operate under syncpoint control, use the MQPMO_SYNCPOINT value in the *Options* field of the MQPMO structure when calling MQPUT. For a get operation, use the MQGMO_SYNCPOINT value in the *Options* field of the MQGMO structure. If not explicitly choosing an option, the default action depends on the platform. The syncpoint control default on OS/390 and Tandem NSK is 'yes'; for all other platforms, it is 'no'.

If a program issues the MQDISC call while uncommitted requests exist, an implicit syncpoint occurs, except on OS/390 batch with RRS, If the program ends abnormally, an implicit backout occurs. On OS/390, an implicit syncpoint occurs if the program ends normally without first calling MQDISC.

For MQSeries for OS/390 programs, use the MQGMO_MARK_SKIP_BACKOUT option to specify that a message should not be backed out if backout occurs (in order to avoid an 'MQGET-error-backout' loop).

7.1 Syncpoints in MQSeries for Windows NT, MQSeries on UNIX systems

Syncpoint support operates on two types of units of work: local and global. A *local* unit of work is one in which the only resources updated are those of the MQSeries queue manager. Here syncpoint coordination is provided by the queue manager itself using a single-phase commit procedure.

A *global* unit of work is one in which resources belonging to other resource managers, such as databases, are also updated. MQSeries can coordinate such units of work itself or the units of work can also be coordinated by an external commitment controller such as another transaction manager.

For full integrity, a two-phase commit procedure must be used. Two-phase commit can be provided by XA-compliant transaction managers and databases such as IBM's TXSeries and UDB. MQSeries Version 5 products (except MQSeries for OS/390) can coordinate global units of work using a two-phase commit process.

7.1.1 Local units of work

Units of work that involve only the queue manager are called *local* units of work. Syncpoint coordination is provided by the queue manager itself (internal coordination) using a single-phase commit process. To start a local unit of work, the application issues MQGET, MQPUT, or MQPUT1 requests specifying the appropriate syncpoint option. The unit of work is committed using MQCMIT or rolled back using MQBACK. However, the unit of work also ends when the connection between the application and the queue manager is broken, whether intentionally or unintentionally.

If an application disconnects (MQDISC) from a queue manager while a unit of work is still active, the unit of work is committed. If, however, the application terminates without disconnecting, the unit of work is rolled back as the application is deemed to have terminated abnormally.

7.1.2 Global units of work

Use global units of work when needing to include updates to resources belonging to other resource managers. Here the coordination may be internal or external to the queue manager:

7.1.3 Internal syncpoint coordination

Queue manager coordination of global units of work is supported only on MQSeries Version 5 products except for MQSeries for OS/390. It is not supported in a MQSeries client environment. Here, the coordination is performed by MQSeries. To start a global unit of work, the application issues the MQBEGIN call.

As input to the MQBEGIN call, supply the connection handle *(Hconn*), which is returned by the MQCONN or MQCONNX call. This handle represents the connection to the MQSeries queue manager.

Again, the application issues MQGET, MQPUT, or MQPUT1 requests specifying the appropriate syncpoint option. This means that MQBEGIN can be used to initiate a global unit of work that updates local resources, resources belonging to other resource managers, or both. Updates made to resources belonging to other resource managers are made using the API of that resource manager. However, it is not possible to use the MQI to update queues that belong to other queue managers. MQCMIT or MQBACK must be issued before starting further units of
work (local or global).

The global unit of work is committed using MQCMIT; this initiates a two-phase commit of all the resource managers involved in the unit of work. A two-phase commit process is used whereby resource managers (for example, XA-compliant database managers such as DB2, Oracle, and Sybase) are firstly all asked to prepare to commit. If any resource manager signals that it cannot commit, each is asked to back out instead. Alternatively, MQBACK can be used to roll back the updates of all the resource managers.

If an application disconnects (MQDISC) while a global unit of work is still active, the unit of work is committed. If, however, the application terminates without disconnecting, the unit of work is rolled back as the application is deemed to have terminated abnormally. The output from MQBEGIN is a completion code and a reason code. When MQBEGIN is used to start a global unit of work, all the external resource managers that have been configured with the queue manager are included. If there are no participating resource managers (that is, no resource managers have been configured with the queue manager) or one or more resource managers are not available, the call starts a unit of work and completes with a warning.

In these cases, the unit of work should include updates to only those resource managers that were available when the unit of work was started. If one of the resource managers is unable to commit its updates, all of the resource managers are instructed to roll back their updates, and MQCMIT completes with a warning. In unusual circumstances (typically, operator intervention), a MQCMIT call may fail if some resource managers commit their updates but others roll them back; the work is deemed to have completed with a ‘mixed’ outcome. Such occurrences are diagnosed in the error log of the queue manager so remedial action may be taken. A MQCMIT of a global unit of work succeeds if all of the resource managers involved commit their updates. For a description of the MQBEGIN call, see the *MQSeries Application Programming Reference* manual.

7.1.4 External syncpoint coordination

External syncpoint coordination occurs when a syncpoint coordinator other than MQSeries (e.g. CICS, Encina, and Tuxedo) has been selected. MQSeries on a UNIX system or MQSeries for Windows NT will register its interest in the outcome of the unit of work, with the syncpoint coordinator. This happens in order to commit or roll back any uncommitted get or put operations as required. The external syncpoint coordinator determines whether one- or two-phase commitment protocols are provided. When an external coordinator is used MQCMIT, MQBACK, and MQBEGIN may not be issued. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR. The way in which an externally

coordinated unit of work is started is dependent on the programming interface provided by the syncpoint coordinator. An explicit call may, or may not, be required. If an explicit call is required, and the MQPUT call specifying the MQPMO_SYNCPOINT option is specified when a unit of work is not started, the completion code MQRC_SYNCPOINT_NOT_AVAILABLE is returned.

The syncpoint coordinator determines the scope of the unit of work. The state of the connection between the application and the queue manager affects the success or failure of MQI calls that an application issues, not the state of the unit of work. It is, for example, possible for an application to disconnect and reconnect to a queue manager during an active unit of work and perform further MQGET and MQPUT operations inside the same unit of work. This is known as a pending disconnect.

7.1.5 Interfaces to external syncpoint managers

MQSeries on UNIX systems and MQSeries for Windows NT support coordination of transactions by external syncpoint managers which utilize the X/Open XA interface. This support is available only on server configurations. The interface is not available to client applications.

Some XA transaction managers (not CICS on Open Systems or Encina) require that each XA resource manager supply its name. This is the string called name in the XA switch structure. The resource manager for MQSeries on UNIX systems is named “MQSeries_XA_RMI”. For further details on XA interfaces refer to XA documentation *CAE Specification Distributed Transaction Processing: The XA Specificatio*n, published by The Open Group.

In an XA configuration, MQSeries on UNIX systems and MQSeries for Windows NT fulfill the role of an XA Resource Manager. An XA syncpoint coordinator can manage a set of XA Resource Managers, and synchronize the commit or backout of transactions in both Resource Managers.

For a statically-registered resource manager:

1. An application notifies the syncpoint coordinator that it wishes to start a transaction.
2. The syncpoint coordinator issues a call to any resource managers that it knows of, to notify them of the current transaction.
3. The application issues calls to update the resources managed by the resource managers associated with the current transaction.
4. The application requests that the syncpoint coordinator either commits or rolls back the transaction.
5. The syncpoint coordinator issues calls to each resource manager using two-phase commit protocols to complete the transaction as requested. The XA specification requires each Resource Manager to provide a structure called an *XA Switc*h. This structure declares the capabilities of the Resource Manager, and the functions that are to be called by the syncpoint coordinator.

There are two versions of this structure:

MQRMIXASwitch
Static XA resource management

MQRMIXASwitchDynamic
Dynamic XA resource management

The structure is found in the following libraries:
mqmxa.lib
Windows NT XA library for Static resource management
mqmenc.lib
Sun Solaris and Windows NT Encina XA library for Dynamic resource management
libmqmxa.a
UNIX systems XA library (non-threaded) for both Static and Dynamic resource management
libmqmxa_r.a
UNIX systems (except Sun Solaris) XA library (threaded) for both Static and Dynamic resource management. The method that must be used to link them to an XA syncpoint coordinator is defined by the coordinator. Also, consult the documentation provided by that coordinator to determine how to enable MQSeries to cooperate with the XA syncpoint coordinator.

The *xa_info* structure that is passed on any *xa_open* call by the syncpoint coordinator should be the name of the queue manager that is to be administered. This takes the same form as the queue manager name passed to MQCONN or MQCONNX, and may be blank if the default queue manager is to be used.

7.2 MQSeries Syncpoint Calls for OS/390

MQSeries for OS/390 provides the MQCMIT and MQBACK calls. Use these calls in OS/390 batch programs to tell the queue manager that all the MQGET and MQPUT operations since the last syncpoint are to be made permanent (committed) or are to be backed out. To commit and back out changes in other environments:

CICS Use commands such as EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK.

IMS Use the IMS syncpoint facilities, such as the GU (get unique) to the IOPCB, CHKP (checkpoint), and ROLB (rollback) calls.

RRS Use MQCMIT and MQBACK or SRRCMIT and SRRBACK as appropriate.

> Note: SRRCMIT and SRRBACK are ‘native’ RRS commands, and are not MQI calls.

For backward compatibility, the CSQBCMT and CSQBBAK calls are available as synonyms for MQCMIT and MQBACK. These are described fully in the *MQSeries Application Programming Reference* manual.

7.3 MQSeries Syncpoint Calls on Windows NT and UNIX systems

The following products provide the MQCMIT and MQBACK calls:

- MQSeries for Windows NT
- MQSeries on UNIX systems

Use syncpoint calls in programs to tell the queue manager that all the MQGET and MQPUT operations since the last syncpoint are to be made permanent (committed) or are to be backed out. To commit and back out changes in the CICS environment, use commands such as EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK.

8. NEXT STEPS

This document should be viewed as a living document. Updates should be done when a new legacy system has been integrated into the EAI Core Architecture or a decision has been made concerning the areas documented in this enablement guide. As an initial next step, SFA needs to adopt and publish EAI Architecture standards. Second, SFA should address system management of the products making up the EAI infrastructure specifically MQSeries and MQSeries Integrator. In addition, the areas of security, backup and recovery planning and application issues should also be researched and documented as part of the application EAI development effort.

9. APPENDIX A: REFERENCE MATERIAL

For more information on the software and hardware prerequisites for the OS/390, please refer to the "MQSeries for OS/390 v5.2 Program Directory" and the "MQSeries for OS/390 v5.2 Concepts and Planning Guide" books on the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

For more information on WebSphere Application Server prerequisites, please refer to the "MQSeries for Windows NT and 2000 Quick Beginnings" book on the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

For more information on EAI BUS prerequisites, please refer to the "MQSeries for Windows NT and 2000 Quick Beginnings" book on the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

For more information on DLSS prerequisites, please refer to the "MQSeries for Compaq (DIGITAL) OpenVMS System Management" book on the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

For more information on PEPS prerequisites, please refer to the "MQSeries for HP-UX v5.2 Quick Beginnings" book on the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

For more information on BTrade prerequisites, please refer to the "MQSeries for HP-UX v5.2 Quick Beginnings" book on the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa.

For more information on how to customize MQSeries objects for application specific requirements, please refer to the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

For more information on MQSeries application error handling, event monitoring and MQSI error handling, please refer to the following books:
"MQSeries Application Programming Reference"
"MQSeries Event Monitoring"
"MQSeries Integrator Introduction and Planning"
on the IBM website: http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

For more information on managing clusters and developing a custom cluster workload exit, please refer to the "MQSeries Queue Manager Clusters" book on the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manuals

For more information on the MQSeries Integrator Control Center and the MQSeries commands and control commands, please refer to the following books:
"MQSeries Integrator Using the Control Center"
"MQSeries MQSC Command Reference"
"MQSeries Systems Administration"
"MQSeries for Compaq (DIGITAL) OpenVMS System Management"
"MQSeries for OS/390 System Administration Guide"
on the IBM website: http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

For more information on the MQSI configuration manger, please refer to the "MQSeries Integrator Using the Control Center" book on the IBM website:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/

MQSeries Application Programming Guide can be found at:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/ - Latest family books

MQSeries Application Programming Reference can be found at:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/ - Latest family books

MQSeries Application Messaging Interface manual can be found at:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/ - Latest family books

MQSeries Using C++ manual can be found at:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/ - Latest family books

MQSeries Using Java manual can be found at:
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/ - Latest family books

10. APPENDIX B: GLOSSARY

A

active log
: See recovery log.

adapter
: An adapter is an attachment facility (program) that enables applications to access MQSeries services. More specifically an adapter is used isolate an application implementing an interface which manages format conversions and application specific behavior.

alias queue object
: A MQSeries object, the name of which is an alias for a base queue defined to the local queue manager. When an application or a queue manager uses an alias queue, the alias name is resolved and the requested operation is performed on the associated base queue.

alternate user security
: A security feature in which the authority of one user ID can be used by another user ID; for example, to open a MQSeries object.

archive log
: See *recovery log*.

asynchronous messaging
: A method of communication between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *synchronous messaging*.

authorization service
: In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a service that provides authority checking of commands and MQI calls for the user identifier associated with the command or call.

B

bootstrap data set (BSDS)
: A VSAM data set that contains:
  - An inventory of all active and archived log data sets known to MQSeries for OS/390
  - A wrap-around inventory of all recent MQSeries for OS/390 activity

  The BSDS is required if the MQSeries for OS/390 subsystem has to be restarted.

browse
: In message queuing, to use the MQGET call to copy a message without removing it from the queue. See also *get*.

browse cursor
: In message queuing, an indicator used when browsing a queue to identify the message that is next in sequence.

BSDS
: Bootstrap data set.

C

channel

See *message channel*.

channel definition file (CDF)
In MQSeries, a file containing communication channel definitions that associate transmission queues with communication links.

channel event
An event indicating that a channel instance has become available or unavailable. Channel events are generated on the queue managers at both ends of the channel.

checkpoint
A time when significant information is written on the log. Contrast with *syncpoint*. In MQSeries on UNIX systems, the point in time when a data record described in the log is the same as the data record in the queue. Checkpoints are generated automatically and are used during the system restart process.

circular logging
In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the process of keeping all restart data in a ring of log files. Logging fills the first file in the ring and then moves on to the next, until all the files are full. At this point, logging goes back to the first file in the ring and starts again, if the space has been freed or is no longer needed. Circular logging is used during restart recovery, using the log to roll back transactions that were in progress when the system stopped. Contrast with *linear logging*.

client
A run-time component that provides access to queuing services on a server for local user applications. The queues used by the applications reside on the server. See also *MQSeries client*.

client application
An application, running on a workstation and linked to a client, that gives the application access to queuing services on a server.

cluster
A network of queue managers that are logically associated in some way.

command
In MQSeries, an administration instruction that can be carried out by the queue manager.

command server
The MQSeries component that reads commands from the system-command input queue, verifies them, and passes valid commands to the command processor.

connect
To provide a queue manager connection handle, which an application uses on subsequent MQI calls. The connection is made either by the MQCONN call, or automatically by the MQOPEN call.

context
Information about the origin of a message.

context security
In MQSeries, a method of allowing security to be handled such that messages are obliged to carry details of their origins in the message descriptor.

control command
In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a command that can be entered interactively from the operating system command line. Such a command requires only that the MQSeries product be installed; it does not require a special utility or

program to run it.

D

data bag
In the MQAI, a bag that allows you to handle properties (or parameters) of objects.

data conversion interface (DCI)
The MQSeries interface to which customer- or vendor-written programs that convert application data between different machine encodings and CCSIDs must conform. A part of the MQSeries Framework.

DB2
A relational database marketed by IBM. Also known as UDB or Universal Database.

DCI
Data conversion interface.

dead-letter queue (DLQ)
A queue to which a queue manager or application sends messages that it cannot deliver to their correct destination.

dead-letter queue handler
A MQSeries-supplied utility that monitors a dead-letter queue (DLQ) and processes messages on the queue in accordance with a user-written rules table.

distributed queue management (DQM)
In message queuing, the setup and control of message channels to queue managers on other systems.

DLQ
Dead-letter queue.

dual logging
A method of recording MQSeries for OS/390 activity, where each change is recorded on two data sets, so that if a restart is necessary and one data set is unreadable, the other can be used. Contrast with *single logging*.

dynamic queue
A local queue created when a program opens a model queue object. See also *permanent dynamic queue* and *temporary dynamic queue*.

E

EID - Enterprise Integration Domain

One of five domains within IAFeB developed to provide an enterprise-wide scalable framework that allows multiple front-end applications (such as web and call centers) to inter-operate with back-end applications (such as policy administration and claims systems) in an effective and efficient manner.

event data
In an event message, the part of the message data that contains information about the event (such as the queue manager name, and the application that gave rise to the event). See also *event header*.

event message
Contains information (such as the category of event, the name of the application that caused the

event, and queue manager statistics) relating to the origin of an instrumentation event in a network of MQSeries systems.

event queue
The queue onto which the queue manager puts an event message after it detects an event. Each category of event (queue manager, performance, or channel event) has its own event queue.

F

Framework
In MQSeries, a collection of programming interfaces that allow customers or vendors to write programs that extend or replace certain functions provided in MQSeries products. The interfaces are:

- MQSeries data conversion interface (DCI)
- MQSeries message channel interface (MCI)
- MQSeries name service interface (NSI)
- MQSeries security enabling interface (SEI)
- MQSeries trigger monitor interface (TMI)

G

get
In message queuing, to use the MQGET call to remove a message from a queue. See also *browse*.

H

HACMP
High Availability Cluster Multi-Processing - IBM's high availability offering for AIX platforms to provide dynamic fail-over within a cluster of separate AIX systems.

I

IAA
Insurance Application Architecture. Insurance business object model.

IAFeB
Insurance architecture for e-business. Framework of common insurance specific functionality built on top MQSeries and MQSeries Integrator. Used by insurance companies to build eBusiness/integration systems upon.

in-doubt unit of recovery
In MQSeries, the status of a unit of recovery for which a syncpoint has been requested but not yet confirmed.

initiation queue
A local queue on which the queue manager puts trigger messages.

installable services
In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, additional functionality provided as independent components. The installation of each component is optional: in-house or third-party components can be used instead. See also *authorization service*, *name service*, and *user identifier service*.

instrumentation event
A facility that can be used to monitor the operation of queue managers in a network of MQSeries systems. MQSeries provides instrumentation events for monitoring queue manager resource definitions, performance conditions, and channel conditions. Instrumentation events can be used by a user-written reporting mechanism in an administration application that displays the events to a system

operator..

L

LDAP
Lightweight directory access protocol.

linear logging
In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the process of keeping restart data in a sequence of files. New files are added to the sequence as necessary. The space in which the data is written is not reused until the queue manager is restarted. Contrast with *circular logging*.

listener
In MQSeries distributed queuing, a program that monitors for incoming network connections.

local definition
A MQSeries object belonging to a local queue manager.

local definition of a remote queue
A MQSeries object belonging to a local queue manager. This object defines the attributes of a queue that is owned by another queue manager. In addition, it is used for queue-manager aliasing and reply-to-queue aliasing.

local queue
A queue that belongs to the local queue manager. A local queue can contain a list of messages waiting to be processed. Contrast with *remote queue*.

local queue manager
The queue manager to which a program is connected and that provides message queuing services to the program. Queue managers to which a program is not connected are called *remote queue managers*, even if the queue managers are running on the same system as the program.

log
In MQSeries, a file recording the work done by queue managers while the queue managers receive, transmit, and deliver messages. The log file is used to recover in the event of failure.

log control file
In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the file containing information needed to monitor the use of log files (for example, their size and location, and the name of the next available file).

log file
In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a file in which all significant changes to the data controlled by a queue manager are recorded. If the primary log files become full, MQSeries allocates secondary log files.

M

message
In message queuing applications, a communication sent between programs. See also *persistent message* and *nonpersistent message*. In system programming, information intended for the terminal operator or system administrator.

message channel
In distributed message queuing, a mechanism for moving messages from one queue manager to another. A message channel comprises two message channel agents (a sender at one end and a

receiver at the other end) and a communication link. Contrast with *MQI channel*.

message channel agent (MCA)
A program that transmits prepared messages from a transmission queue to a communication link, or from a communication link to a destination queue. See also *message queue interface*.

message channel interface (MCI)
The MQSeries interface to which customer- or vendor-written programs that transmit messages between a MQSeries queue manager and another messaging system must conform. A part of the MQSeries Framework.

message descriptor
Control information describing the message format and presentation that is carried as part of a MQSeries message. The format of the message descriptor is defined by the MQMD structure.

message priority
In MQSeries, an attribute of a message that can affect the order in which messages on a queue are retrieved, and whether a trigger event is generated.

message queue
Synonym for *queue*.

message queue interface (MQI)
The programming interface provided by the MQSeries queue managers. This programming interface allows application programs to access message queuing services.

message queuing
A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

messaging
See *synchronous messaging* and *asynchronous messaging*.

model queue object
A set of queue attributes that act as a template when a program creates a dynamic queue.

MQSeries – Message Queue Series
A family of IBM licensed programs that provides message queuing services across a broad array of operating system platforms and network protocols.

MQSeries Administration Interface (MQAI)
A programming interface to MQSeries.

MQSeries client
Part of a MQSeries product that can be installed on a system without installing the full queue manager. The MQSeries client accepts MQI calls from applications and communicates with a queue manager on a server system.

MQSeries commands (MQSC)
Human readable commands, uniform across all platforms, that are used to manipulate MQSeries objects. Contrast with *programmable command format (PCF)*.

MQSI - MQSeries Integrator
Second generation message broker product the provides basic message routing and data translation capabilities.

MQWF
MQSeries Workflow. A workflow product built to executed long running transactions and other

workflow functions over a MQSeries foundation.

N

namelist
A MQSeries object that contains a list of names, for example, queue names.

name service
In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the facility that determines which queue manager owns a specified queue.

name service interface (NSI)
The MQSeries interface to which customer- or vendor-written programs that resolve queue-name ownership must conform. A part of the MQSeries Framework.

nonpersistent message
A message that does not survive a restart of the queue manager. Contrast with *persistent message*.

O

object
In MQSeries, an object is a queue manager, a queue, a process definition, a channel, a namelist, or a storage class (OS/390 only).

object authority manager (OAM)
In MQSeries on UNIX systems, MQSeries for AS/400, and MQSeries for Windows NT, the default authorization service for command and object management. The OAM can be replaced by, or run in combination with, a customer-supplied security service.

output log-buffer
In MQSeries for OS/390, a buffer that holds recovery log records.

P

page set
A VSAM data set used when MQSeries for OS/390 moves data (for example, queues and messages) from buffers in main storage to permanent backing storage (DASD).

performance event
A category of event indicating that a limit condition has occurred.

persistent message
A message that survives a restart of the queue manager. Contrast with *nonpersistent message*.

platform
In MQSeries, the operating system under which a queue manager is running.

point of recovery
In MQSeries for OS/390, the term used to describe a set of backup copies of MQSeries for OS/390 page sets and the corresponding log data sets required to recover these page sets. These backup copies provide a potential restart point in the event of page set loss (for example, page set I/O error).

principal
In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a term used for a user identifier. Used by the object authority manager for checking authorizations to system resources.

process definition object
A MQSeries object that contains the definition of a MQSeries application. For example, a queue

manager uses the definition when it works with trigger messages.

programmable command format (PCF)
A type of MQSeries message used by:
- User administration applications, to put PCF commands onto the system command input queue of a specified queue manager
- User administration applications, to get the results of a PCF command from a specified queue manager
- A queue manager, as a notification that an event has occurred

Contrast with *MQSC*.

Q

queue
A MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed.

queue manager
A system program that provides queuing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. See also *local queue manager* and *remote queue manager*. A MQSeries object that defines the attributes of a particular queue manager.

queuing
See message queuing.

R

recovery log
In MQSeries for OS/390, data sets containing information needed to recover messages, queues, and the MQSeries subsystem. MQSeries for OS/390 writes each record to a data set called the *active log*. When the active log is full, its contents are off-loaded to a DASD or tape data set called the *archive log*. Synonymous with *log*.

remote queue
A queue belonging to a remote queue manager. Programs can put messages on remote queues, but cannot get messages from remote queues. Contrast with *local queue*.

remote queue manager
To a program, a queue manager that is not the one to which the program is connected.

remote queue object
See *local definition of a remote queue*.

remote queuing
In message queuing, the provision of services to enable applications to put messages on queues belonging to other queue managers.

reply message
A type of message used for replies to request messages.

request message
A type of message used to request a reply from another program.

RESLEVEL
In MQSeries for OS/390, an option that controls the number of CICS user IDs checked for API-resource security in MQSeries for OS/390.

return codes
The collective name for completion codes and reason codes.

S

security enabling interface (SEI)
The MQSeries interface to which customer- or vendor-written programs that check authorization, supply a user identifier, or perform authentication must conform. A part of the MQSeries Framework.

server
(1) In MQSeries, a queue manager that provides queue services to client applications running on a remote workstation. (2) The program that responds to requests for information in the particular two-program, information-flow model of client/server. See also *client*.

signaling
In MQSeries for OS/390 and MQSeries for Windows 2.1, a feature that allows the operating system to notify a program when an expected message arrives on a queue.

single logging
A method of recording MQSeries for OS/390 activity where each change is recorded on one data set only. Contrast with *dual logging*.

synchronous messaging
A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

system.command.input queue
A local queue on which application programs can put MQSeries commands. The commands are retrieved from the queue by the command server, which validates them and passes them to the command processor to be run.

T

thread
In MQSeries, the lowest level of parallel execution available on an operating system platform.

trace
In MQSeries, a facility for recording MQSeries activity. The destinations for trace entries can include GTF and the system management facility (SMF). See also *global trace* and *performance trace*.

transmission queue
A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

trigger event
An event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

triggering
In MQSeries, a facility allowing a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

trigger message
A message containing information about the program that a trigger monitor is to start.

trigger monitor
A continuously-running application serving one or more initiation queues. When a trigger message

arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

trigger monitor interface (TMI)
The MQSeries interface to which customer- or vendor-written trigger monitor programs must conform. A part of the MQSeries Framework.

U

undelivered-message queue
See dead-letter queue.

unit of recovery
A recoverable sequence of operations within a single resource manager. Contrast with *unit of work*.

unit of work
A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or after a user-requested syncpoint. It ends either at a user-requested syncpoint or at the end of a transaction. Contrast with *unit of recovery*.

user identifier service (UIS)
In MQSeries for OS/2 Warp, the facility that allows MQI applications to associate a user ID, other than the default user ID, with MQSeries messages.

utility
In MQSeries, a supplied set of programs that provide the system operator or system administrator with facilities in addition to those provided by the MQSeries commands. Some utilities invoke more than one function.

X

XML
Extensible markup language