WebSphere® MQ Integrator Enabler

Development Guide

NOTE:

Before using this information and the product it supports, read the information in *Notices* on page 158.

Fifth Edition (June 2002)

Printed in USA.

Contents

Figures

About this book

This publication contains information on how to customize WebSphere MQ Integrator Enabler (WMQI Enabler). It is designed to give instructions on how to modify WMQI Enabler as it is shipped with the Model Office, to meet the needs of a particular implementation. Alternatively, this publication can be used in the development of an enterprise-wide installation of WMQI Enabler.

Who should read this book

Developers who will be working to customize the WMQI Enabler product for their requirements, architects who are planning an implementation of WMQI Enabler, and anyone working with XML messaging.

Terminology used in this book

All new terms introduced in this book are defined in the *Glossary*.

This book uses the following shortened names:

- MQSeries®: a general term for IBM MQSeries messaging products.
- DB2®: a general term to encompass IBM DB2 Universal Database® Enterprise Edition, Connect Enterprise Edition, and Extended Enterprise Edition.
- LDAP: refers to the directory structure as characterized by the Lightweight Directory Access Protocol.

Prerequisite and related information

It is recommended that readers become familiar with the WMQI Enabler architecture by reading the WMQI Enabler ***Technical Architecture Book***.

It is not necessary for one person to have all of these skills. These skills can be represented as a team.

Before beginning work to customize WMQI Enabler, it is recommend that business analysts and project managers read through the WMQI Enabler ***Planning Guide*** in order to provide input to developers on their implementation requirements.

The following publications contain other related information about WebSphere MQ Integrator Enabler:

- ***Installation and Setup Guide***
- ***Application Integration Guide***
- ***Model Office Reference Manual***

The following other publication is used for information about the prerequisite products:

- ***Business Integration Solutions with MQSeries Integrator*** (IBM Redbook).

How to get additional information

Visit the following home page at:

http://www.ibm.com/software/mqseries/support/

By following this link you can find:

- The latest information about MQSeries family of products.
- Download Support Packs.
- Access FAQs.
- Access MQSeries family publications library.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments or suggestions about this book or any other *WebSphere MQ Integrator Enabler* documentation:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom

- By fax:

  - From outside the U.K., after your international access code use 44-1962-816151
  - From within the U.K., use 01962-816151

- Electronically, use the appropriate network ID:

  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address / telephone number / fax number / network ID

Chapter 1
Introduction

WebSphere MQ Integrator Enabler (WMQI Enabler) is a starting point for integrating applications in an enterprise. Undoubtedly, an implementation will require various customizations to mold or extend WMQI Enabler to meet specific needs and business requirements. This guide for developers takes a “how to” approach, including topics for possible customizations and development work. Each chapter is based on understanding the modifications necessary for a particular task or area of the WMQI Enabler implementation.

Both MQSeries Integrator (MQSI) and MQSeries Workflow (MQSWF) are given a chapter in this publication that describes possible customization options. It is important to recognize that the development of customized MQSWF process flows is required for the deployment of WMQI Enabler, if complex process definitions are required for a particular business implementation. MQSI message flows provide the functionality that drives WMQI Enabler. The existing MQSI message flow functionality is designed to generically support all WMQI Enabler messages, and as a result does not require additional customizations. However, there are a few instances where customization of some MQSI message flows may be desirable and possible to extend existing WMQI Enabler functionality. Some opportunities for such modifications are indicated in this document.

Chapter 2
Adding a new system

New systems will need to be integrated with WebSphere MQ Integrator Enabler (WMQI Enabler) to allow for processing of transactions through WMQI Enabler to other systems already integrated with WMQI Enabler. An adapter will have to be created to act as a translator between the new system and WMQI Enabler. The WMQI Enabler ***Application Integration Guide*** contains information on what requirements an adapter must satisfy to allow an application to integrate with WMQI Enabler. Once the adapter is ready to be deployed to the WMQI Enabler implementation, there will be a few steps necessary for WMQI Enabler to recognize the new application. The following sections will describe the steps that are necessary, such as adding an entry for adapters in the WMQI Enabler Symbolic Destination Resolution (SDR) table.

System profile tables

A system Profile consists of the following three tables:

SYSTEM_STATUS_TABLE

SYSTEM_STORE_FLAG_TABLE

SYSTEM_BACKUP_TABLE

System status table

All systems that require interaction with WMQI Enabler must have an entry in the SYSTEM_STATUS_TABLE. This allows WMQI Enabler to keep up to date with the availability of each system. "LANGUAGE" defines the language associated with the NLS handling of a particular system.
"BLOCKED_BY_SYS_INTERACTION_FLG" indicates if a process using the system is performing a system interaction check. "SYS_ACTIVE" indicates whether or not the system is active. "SYS_REQUESTED_SHUTDOWN_FLG" indicates if the system has requested to shut down. "ERROR_MSG_DEST" indicates default error processing and allows the system to specify if it would like error messages related to this system to be returned to the system, the default error queue, both, or neither. Related values are SOURCE, DEFAULT, BOTH, and NONE. “RETURN_SUCCESS_CODE” indicates the success code indicated in a response message required by this system.
“RETURN_FAILURE_CODE” indicates a failure code required by this system.
"SYS_SYMBOLIC" is the symbolic representing the system.

Note: "RETURN_SUCCESS_CODE", when it exists, populates the Feedback field of the MQMD header. "RETURN_FAILURE_CODE", though it exists as a placeholder for customization, is not currently implemented.

This table can be populated and updated by using the hub only commands SystemRestart and UpdateSystemProfile. SystemRestart will provide a default profile if one does not already exist for that system. If a profile does exist, SystemRestart will reset the "SYS_ACTIVE", "BLOCKED_BY_SYS_INTERACTION_FLG", and "SYS_RQUESTED_SHUTDOWN_FLG" to indicated that the system is active and does not wish to shut down. The UpdateSystemProfile command provides the means for setting the language, error message destination, and the return codes.

A sample SYSTEM_STATUS_TABLE is shown in the following figure:

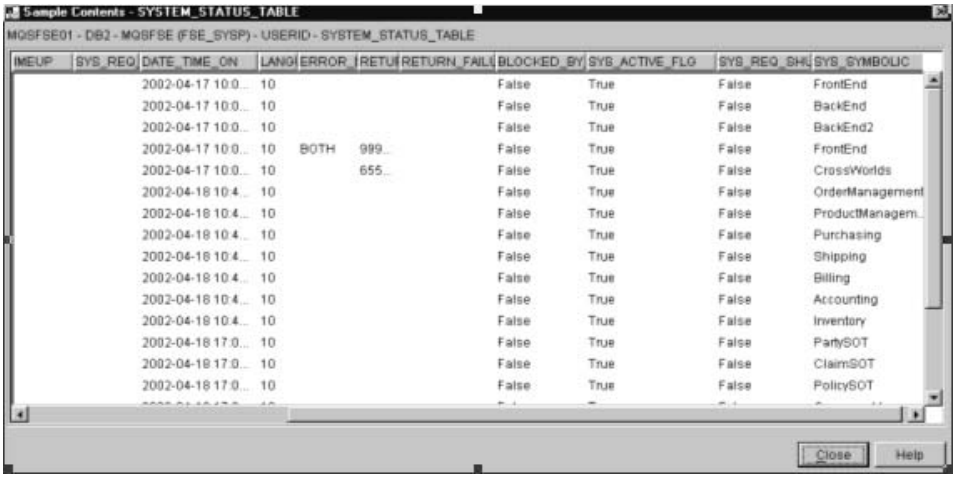


Figure 1: Sample SYSTEM_STATUS_TABLE.

System Store Flag Table

The SYSTEM_STORE_FLAG_TABLE allows the system to indicate if it wants to store a message if it has failed a system interaction check. If the system does store the message, it is reprocessed once the destination system is restarted.

This table can be populated and updated using the hub only command UpdateSystemProfile.

A sample SYSTEM_STORE_FLAG_TABLE is shown in the following figure:

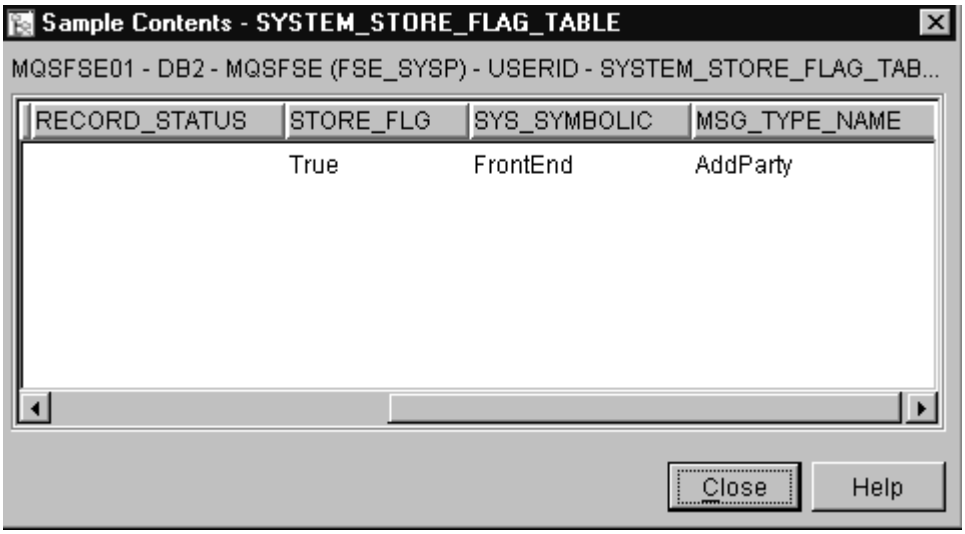


Figure 2: Sample SYSTEM_STORE_FLAG_TABLE.

System backup table

The SYSTEM_BACKUP_TABLE contains a linked structure of records that indicate back up systems. If a system is down when a system interaction check is performed, the status of its backup, indicated here, is checked. If that backup is down, then its back up is checked, and so on until an active system is found, or no more backup systems are indicated. If a backup is found to be active, the message is sent there.

This table can be populated and updated using the hub only command UpdateSystemProfile.

A sample SYSTEM_BACKUP_TABLE is shown in the following figure:

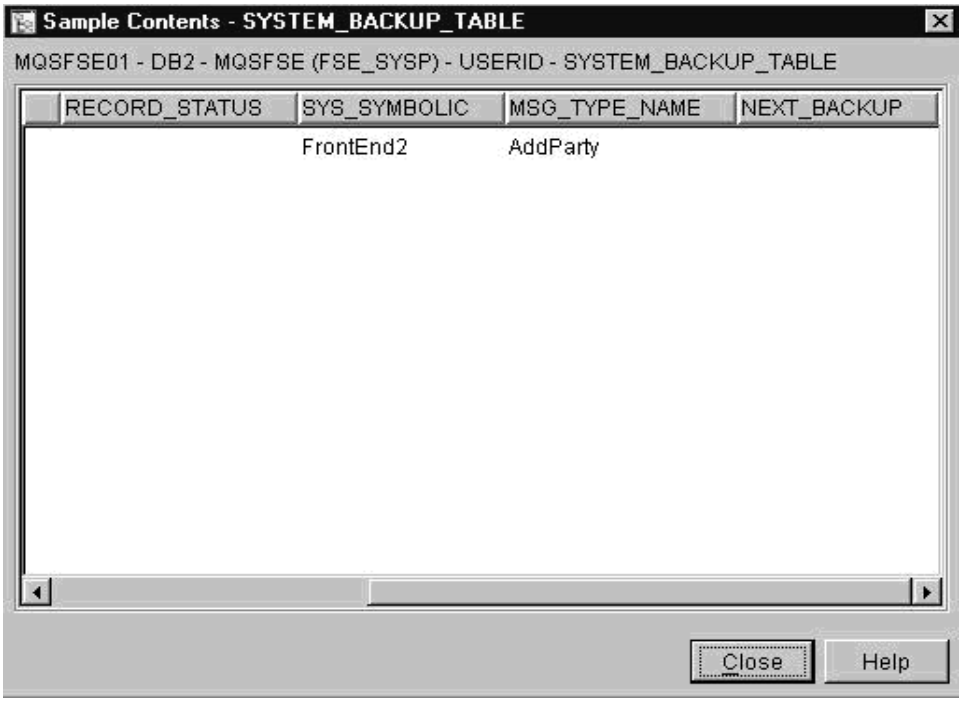


Figure 3: Sample SYSTEM_BACKUP_TABLE.

Putting It Together

For example, a FrontEnd system may indicate that it wants to perform a system interaction check, and that it requires communications with a BackEnd system. The record in the System_Status_Table for the BackEnd is examined to determine if the BackEnd is currently active and not requesting to shutdown. If the BackEnd is not active or about to shutdown, the System_Backup_Table is checked to see if the BackEnd has a back up system. If it does, the back up system is examined to see if it is up. Ultimately, if the BackEnd is not available, and none of its back up systems are available, the store forward flag of the FrontEnd system is examined in the System_Store_Flag_Table. If the flag is "True", then the message is stored to a database. When the BackEnd system issues a SystemRestart command, the message is pulled from the database and reprocessed. If the store forward flag for the FrontEnd system is "False" the request message is discarded, and the FrontEnd receives an error message indicating that system interaction could not find an available system.

Symbolic destination resolution table

SDR explained

This section describes how to integrate a system with WMQI Enabler by adding an entry in the Symbolic Destination Resolution (SDR) table. This table is created with the installation of the WMQI Enabler product and used by the Model Office. Alternatively, this table can be created in DB2 through the DB2 Control Center or Command Center, or by using SQL statements. There are 3 columns in the SDR table, containing the following fields, as shown in the figure below.

SYS_SYMBOLIC (character)

Q_NAME (character)

Q_MGR_NAME (character)

The symbolic name of the system corresponds to the Source Logical ID and Destination Logical ID in the WMQI Enabler XML message header. The MQSeries queue designated by the "Q_NAME" is the destination for messages intended for the associated system.

A sample SDR_TABLE is shown in the following figure:

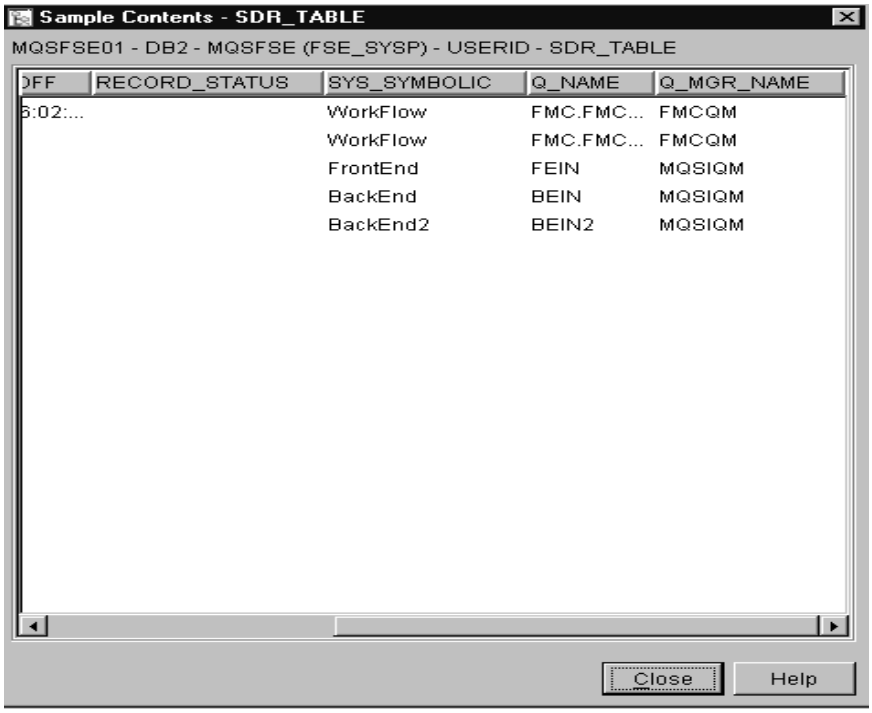


Figure 4: SDR table.

SDR duties

The SDR function is used to resolve the logical to physical system identification needs of the message. The MQSI process flow "SDR" is used to implement this function.

Request messages

On a request XML message from a front-end source system, the Symbolic Destination Resolution table is used to find the destination queue for the back-end system that is the intended receiver of the XML message. The destination system symbolic can be specified in multiple ways to allow for multiple scenarios. If the destinationLogicalId attribute of the message is populated, then it is used to identify the SYS_SYMBOLIC. If the destinationLogicalId is not populated, the DEFAULT_DESTINATION_SYMBOLIC associated with this message type as specified in the Message_Profile_Table is used to identify the SYS_SYMBOLIC. In the event that system interaction is required by a message, the system symbolics indicated in the System_Interaction_Table associated with the given message are used to identify the destinations of the request message. In this case, the destinationLogicalId and the DEFAULT_DESTINATION_SYMBOLIC are ignored. The "SYS_SYMBOLIC" name is used to select the "Q_NAME" and "Q_MGR_NAME" in the SDR table. The system symbolic name in the sourceLogicalId attribute of the message is stored in the Process_State_Table. Also, information stored in the ReplyQ and ReplyQMgr fields of the MQMD header of the request message are stored in the Process_State_Table. Request XML messages can also originate from back-end source systems.

Response messages

On a response XML message from the destination system, when the message is leaving the MQSI portion of WMQI Enabler, the SDR table identifies the queue where the message will be placed. If the reply information of the original request was available and saved in the Process_State_Table, it is used as the destination of the response message. In that case, the queue and queue manager are already specified and the SDR processing is not required. If the reply information was not available, the Source Logical Id that was stored when the original request message entered WMQI Enabler is used with the SDR table to route the response message. If that information does not exist, the Destination Logical Id in the response message is used for routing.

There could be two different entries with similar symbolic names. For instance, if a business implementation is using two policy administration systems, there might be two entries, with the symbolic names being "POLICY1" and "POLICY2". Keep in mind that entries in the Symbolic Destination Resolution table are case sensitive and must be precise.

Adding entries to the SDR table

Allowing a new system to be accessible within WMQI Enabler entails adding an entry for the system in the SDR table. Entries in the SDR table can be added, using any preferred DB2 method. WMQI Enabler provides a GUI configuration tool called the WMQI Enabler Configuration Utility for loading entries as well. Additionally, WMQI Enabler supports an XML-based Hub command called UpdateSDREntry that can be used to add a record to the table, as well as replacing existing records with updated information. Anytime a change is made to an MQSeries Queue Name or Queue Manager, it will be important to ensure that the SDR is updated accordingly.

For information on how to work with a DB2 database, see ***IBM DB2 Universal Database for Windows NT:*** *Quick Beginnings Version 6, Appendix A.*

Setting up the queues in MQSeries

There are two methods for setting up the MQSeries queues.

Method 1: All of the queues are local, and belong to the same Queue Manager.

Method 2: Channels are setup between multiple Queue Managers.

For method 1, there is one queue per entry/exit point to the WMQI Enabler portion on the MQSI Queue Manager. These queues are channel queues. The number of queues setup should be equal to the number of entry/exit points created for the various systems.

Method 2 uses at least one (possibly several) additional queue managers, which are in the same MQSeries cluster. Two channels will need to be setup for each additional Queue Manager along with a transmission queue, one being a Sender Channel and one a Receiver Channel.

The name of the queue that MQSeries Workflow uses is called: FMC.FMCGRP.EXE.XML. It is an Alias and is used as a dynamic queue.

Additional work will be necessary to implement an adapter for each new system that is planned to be integrated with WMQI Enabler. In addition, it may be necessary to create an MQSeries Workflow process flow to supervise making a call to the system associated with this new adapter depending on whether the adapter is used as a part of a complex or long running message integration scenario.

Keep in mind that the implementation architecture (i.e., number of servers in the enterprise, queue manager topology, MQSI configuration, MQSWF configuration, etc.) have an effect on how the MQSeries portion of WMQI Enabler will need to be configured. More information on this topic can be found in the WMQI Enabler ***Technical Architecture Book***. The MQSeries product documentation should also be used for a complete reference on how to implement the WMQI Enabler integration topology.

Adding entries to the CRF

By design, once a complete WMQI Enabler solution is up and running, the Cross Reference Function is an optional feature where all functionality of the Cross Reference Function (CRF) will occur automatically if desired. However, at the onset of an WMQI Enabler implementation, it is necessary to populate the CRF table with information that describes the existing operational environment. This effort generally entails a significant conversion effort that is based on analyzing existing systems to determine a normalized view of what systems use automic data as defined by the XML integration vocabulary, and what systems actually own the data.

Entries can be added to the CRF table by creating an XML message with the appropriate WMQI Enabler header entries that represents the state of the data that is being processed through WMQI Enabler. To ensure data integrity, it is recommended that entries not be added manually via database tools to the CRF table database. However, WMQI Enabler supports hub only messages designed to maintain the CRF table. These messages can be used by adapters to supplement the CRF processing normally accomplished using information included as a part of message oriented integration, to do standard add, update, deleted, and modify activities against CRF entries, When publish and subscribe is used within WMQI Enabler, the hub CRF messages are required by the subscriber adapters as a way of synchronizing the CRF with the results of the internal application processing triggered by receipt of a message via subscription.

Chapter 3
Removing a system

If it is necessary to discontinue interaction of a system with WMQI Enabler, the procedures described here can be taken to ensure that the system will no longer be able to send and receive messages through WMQI Enabler.

System profile tables

The system can be deactivated by issuing a hub only command of SystemShutdown. Once the requesting system is no longer in use by WMQI Enabler processes, it will indicate that it has shutdown, and messages using system interaction will not be allowed to access that system.

Message profile

The message profile contains metadata describing how a specific type of message should be processed within the hub. The MQSI_MSG_ENABLED_FLG in the Message_Profile_Table can be set to "False", and that message type will not be processed through the hub. The UpdateMessageProfile message can be used to set this flag and deactivate definitions for messages that are no longer necessary as a result of removing systems from the WMQI Enabler integration environment. Also, the record for the desired Message Profile can be deactivated by setting the DATE_TIME_OFF field.

Symbolic destination resolution

The corresponding entry in the Symbolic Destination Resolution (SDR) table for the system that is planned to be removed should be marked as inactive. A record is considered inactive when it's DATE_TIME_OFF field is populated. This method is preferred, rather than completely deleting the entry.

MQSeries queues

If there is a specific queue that corresponds to the system to be removed, it should be taken out of use by using the appropriate MQSeries commands.

Cross reference function

It is likely that there will be multiple entries in the Cross Reference Function (CRF) table corresponding to associations with various data elements in other systems.To maintain data integrity, these entries should be marked as inactive. A record is considered inactive when it's DATE_TIME_OFF field is populated.

Chapter 4
XML language and message profile

Within the WMQI Enabler product, XML is utilized as the messaging medium for the conveyance of data from source, through WMQI Enabler, to the target.

One of the guiding principles of WMQI Enabler is to allow support for industry specific XML languages and architectures. Examples of this architecture include the retail and manufacturing industry which uses the OAG XML messaging architecture, the banking industry which uses the Interactive Financial eXchange (IFX) messaging architecture, and the insurance industry which uses the IAA-XML messaging architecture. All of the XML messaging architectures are flexible, open, and customizable to fit the needs of the business scenarios being implemented.

WMQI Enabler demonstrates two approaches to implementing XML messages. The first approach is to remodel industry content in the context of the WMQI Enabler Interface Design Model (IDM). This approach is a modeling exercise that extends the model as a means of conveying integration semantics through extension and specialization of a data model. Regardless of the source of the content (IAA, IFX, OAG, etc.) the messages generated are in the architectural form of the IDM. IDM messages are useful for internal integration projects where customized messages yield an optimized integration environment.

The alternative approach is an XML messaging technique that combines the WMQI Enabler header and an existing XML message architecture, as described below.

XML messaging technique

The "wrappering" technique used to apply the WMQI Enabler header to the messaging architectures used within this product can be found in the Industry Reference Manuals. The concept is that the WMQI Enabler header contains a "command container" that is used to "wrap" or enclose the XML message associated with a specific XML message dialect. The XML message is then built using a customized DTD that represents the specified architecture and WMQI Enabler header.

The figure below shows a graphical representation of the WMQI Enabler "wrappering" method using an IFX messaging architecture is illustrated:

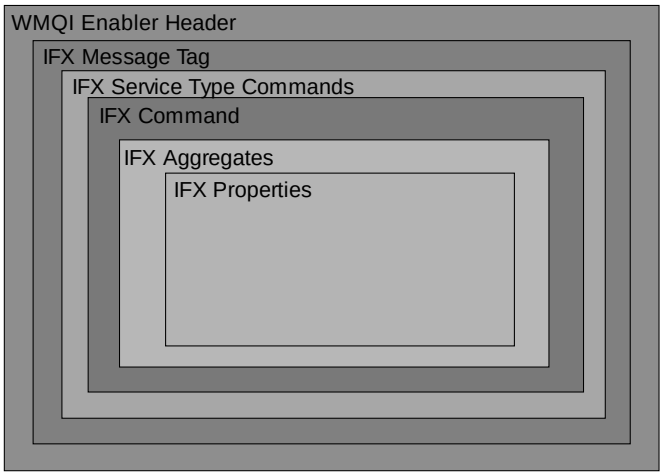


Figure 5: IFX structure.

The advantage of this approach is that it allows applications that support existing standards to leverage WMQI Enabler functionality even if the message set they are using does not have the required semantics. In this case, the wrappering process is used to add the header as messages enter the hub and then remove the header as messages leave the hub, so that the applications do not have to be made hub aware.

For information about a specific industry, please refer to that Industry Reference Manual or the industry manual that is the closest match for the industry being implemented.

Alternative XML within data model

From an WMQI Enabler perspective, there are two types of XML vocabularies:

- Internal
- External

The external vocabulary is described above and these options are further explored in the ***Technical Architecture Book***. Internal vocabularies are generally data model-based and are used to develop custom message sets that can be optimized to the specific requirements of the applications being integrated.

WMQI Enabler header

The WMQI Enabler message header contains all of the information necessary to route information from a source system to a target system. The adapter must be able to insert the necessary information to have a message routed correctly. The fields are completed based on the business scenario and message architecture being used. The following fields must have entries in order to correctly process a message:

authenticationId: Used for authorizing access to specific functions on a target system. This value may be different than the logon id used to access web applications or other systems. Authentication Id is used by the WMQI Enabler Logon and Logoff commands to activate/deactivate a session for messages requiring session validation.

sourceLogicalId: Is the symbolic name used to represent the source application initiating the request message. This naming is necessary for the target system to make sure that the response message is routed correctly back to its sender.

destinationLogicalId: Is the symbolic name used to represent the target application to which messages are routed.

sessionId: This is the session id supplied by WMQI Enabler when a Logon command is processed. It is used during WMQI Enabler session validation.

bodyType: This field is used to identify the type of message to be processed. For example: WorkFlow indicates that this message was generated by WorkFlow, HUBONLYONLINE indicates this is a hub only online message, and IAA-XML indicates this is a message with an IAA structure.

publish: This field indicates whether or not publishing is desired for this message. This field can be overridden by the message profile.

bodyCategory: Used for the specific message type. Also used as the default workflow process name. More specifically, it allows WMQI Enabler to determine the type of message, or command, without parsing actual business content. This entry ties to the Message Profile metadata entries.

CommandReference: Holds a reference to the command id to assist in keeping track of multiple commands within one message. There is one command reference per CrfActionGroup.

keyGroupType: The aggregate name used within the business content portion of the message. This field is an attribute for the KeyGroup aggregate.

AlternateId: Holds all valid hub information for CRF implementation, the source system, and necessary processing requests. It is possible to have one to many associations per KeyGroup. Typically related to the number of occurrences of the keyGroupType.

COMMAND: Symbolizes the container used by WMQI Enabler to wrapper the business content for a specific messaging architecture.

Adding the WMQI Enabler header to an existing DTD

XML is an extensible language that allows the user to define business specific data. The data is defined within a Data Type Definition (DTD) to assist the user in creating XML messages for its specific purpose. The WMQI Enabler header is provided as a text document for easy manipulation. When applying the WMQI Enabler header to an existing DTD, you can use any text editor of preference. The following steps are necessary to add the WMQI Enabler header to an existing DTD.

1. Open the Header.txt document. [check filename]
2. Select the entire document and copy to the clipboard.
3. Open the existing DTD in any text editor or XML tool.
   (Make sure that you are using the text interface if you are using a graphical tool).
4. Paste the contents of the clipboard to the beginning of the DTD.
   (after the XML tag).
5. Within the Element definition for the COMMAND tag, associate the root of the existing DTD to this element (<!ELEMENT COMMAND (IFX)>).

   This will allow the header information to stop and signify the beginning of the business content.
6. Check the DTD for errors.
7. Check for conformity and a valid DTD.

In the case of an original DTD being created, it is possible to open the **Header.txt** document. Further business specific content can be added, the DTD can be saved for it's business purpose.

The following sample represents XML generated after the addition of the WMQI Enabler header:

```
<?xml version="1.0" encoding="UTF-8"?>
<?ifx version="1.0.1" newfileuid="some GUID"?>
<!DOCTYPE IFX PUBLIC "-//ifx forum//ifx 1.0.1 dtd//en"
http://www.ifxforum.org/dtd/ifx1.0.1.dtd>
<!DOCTYPE Message SYSTEM "MQSFSE001222.dtd">
<Message id="M5441920" sessionId="2914320" version="1.4"
bodyType="IAA-XML" timeStampCreated="2000-10-22-08.00.00"
sourceLogicalId="FrontEnd" destinationLogicalId="Party"
authenticationId="SysAdmin" crfCmdMode="alwaysRespond"
publish="true">
<ErrorInfo>
     <errorMessageType/>
     <errorcode/>
     <errorMessageText/>
                  <errorState/>
</ErrorInfo>
<Default>
     <DefaultCurrency>USD</DefaultCurrency>
</Default>
<CrfActionGroup bodyCategory="AddPartyRequest"
crfPublish="true" crfCmdMode="alwaysRespond"
destinationLogicalId="Party">
```

```
<CommandReference refid="CMD1"/>
     <KeyGroup id="K1" keyGroupType="Person">
           <AlternateId value="123450005"
 sourceLogicalId="FrontEnd" state="referenced"/>
     </KeyGroup>
     <KeyGroup id="K2" keyGroupType="ContactPreference">
           <AlternateId value="123450005"
 sourceLogicalId="FrontEnd" state="referenced"/>
                 </KeyGroup>
</CrfActionGroup>
<COMMAND>
<IFX>
     <SignonRq>
     …
     </SignonRq>
     <PaySvcRq>
           <RqUID>some GUID</RqUID>
           <SPName>Avolent.com</SPName>
           <PmtAddRq>
                 <RqUID>some GUID</RqUID>
                 <AsyncRqUID>some GUID</AsyncRqUID>
                 <PmtInfo>
                 …
                 </PmtInfo>
           </PmtAddRq>
     </PaySvcRq>
</IFX>
</COMMAND>
<Message>
```

The DOCTYPE path can be eliminated by placing the DTD and the XML messages in the same directory.

Message profile requirements

All messages entering WMQI Enabler require a valid message profile in the FSE_MSGP database. A script is provided to populate the database with message profiles for each Hub Only command. The profiles indicate that the MessageType is enabled and each validation check is not required. Each profile has the following fields.

Required fields

MessageTypeName
This is the message type that exists in the bodyCategory of CRFActionGroup or cmdType of hub only commands. For example, ‘AddPerson’ or ‘Logon’. This is the field used to pull the profile out of the database. This field is also the default name of the Workflow process associated with the message.

MQSISessionValidationFlag
Specifies if session validation processing is required. Valid values are ‘True’ and ‘False’. Logon and Logoff commands will bypass session processing whether the flag in their profile indicates it or not. Session validation confirms that the sessionId in the message header is valid within the hub database. A valid session id is generated by issuing a logon command.

MQSIMessageSequenceValidationFlag
Specifies if sequence validation processing is required. Valid values are 'True’ and 'False’. This flag can designate the order in which messages are to be processed. Care should be taken to insure that the initial message does not require a prior message or processing will be prohibited.

MQSISystemInteractionCheckFlag
Specifies if system interaction processing is required. Valid values are 'True’ and 'False’. If system interaction is specified, the status of the systems in the SystemInteractionList are checked. If required systems are down with no backup systems available, and store forward is not specified by the requesting system, an error message is returned indicating that there were system interaction problems for the down systems.

WorkFlowManagementFlag
Specifies if the message requires Workflow processing. Valid values are 'True’ and 'False’. The Workflow supporting fields are WorkFlowQueueManager, WorkFlowQueue, WorkFlowDataStructureName, WorkFlowSymbolic, WorkFlowReplyToQueueManager, WorkFlowReplyToQueue,

WorkFlowProcessName, and the WorkflowParameterList. A valid Workflow process must exist for this message. It can be the same as the message type name or, if it exists, specified in the WorkFlowProcessName.

TraceFlag
Specifies if a trace dump is desired when using the TraceLog subflow. Valid values are 'True' and 'False'. When specified as true, the trace information in the WorkArea is written to the Trace_Table.

Optional fields

PublishFlag
Specifies if publishing is required. Valid values are 'True' and 'False'. This flag is utilized to indicate that the message should be forwarded to subscribed systems in a non-error situation.

OverrideFlag
The override flag indicates whether the publish attribute in the message header can be used to specify publishing. Valid values are 'True' and 'False'.

PublishErrorFlag
Specifies if publishing upon error is required. Valid values are 'True' and 'False'. This flag is utilized to indicate that the message should be forwarded to subscribed systems when an error occurs in this message type.

WorkFlowQueueManager
If Workflow processing is required, this field indicates the queue manager for Workflow. For example, FMCQM.

WorkFlowQueue
If Workflow processing is required, this field indicates the queue for Workflow For example, FMC.FMCGRP.EXE.XML.

WorkFlowDataStructureName
This field indicates the data structure name that workflow expects for it's initial message. For example, ProcessTemplateExecute

WorkFlowSymbolic
This field indicates the symbolic that specifies Workflow in the SDR table. This field is used in processing if the WorkFlowQueue is not specified. For example, Workflow.

WorkFlowReplyToQueueManager
This field specifies the queue manager that Workflow is to reply to. For example, MQSIQM.

WorkFlowReplyToQueue
This field specifies the queue that Workflow is to reply to. For example, MQWF_END.

WorkFlowProcessName
This field specifies the Workflow process name associated with this message type. If this field is not present, the message type is used. If this field exists, a valid Workflow process of the same name must exist also.

HubQueueManager
This field specifies the queue manager that the Hub uses. For example, MQSIQM.

UseHubQMgrAsReplyFlag
This flag indicates whether or not response messages should be sent back to the queue manager specified in the ReplyTo information of the MQMD header, or the HubQueueManager in the profile. If this field does not exist in the update message, it is defaulted to 'False', indicating that the ReplyTo information should be used.

PublishTopic
This field is used to specify the topic that this message is to be considered as when being published. If this field is null when the UpdateMessageProfile command is processed, the value of the MessageTypeName is used to populate the column in the database.

DefaultDestinationSymbolic
This field specifies the default destination symbolic that is used when this message type does not have a destinationLogicalId. For example, BackEnd

MessageTypeDependency
This field indicates the dependency used in sequence validation. For example, AddPerson could be a dependency for ModifyPerson.

MQSIMessageEnabledFlag
This flag indicates if this message type is allowed for hub processing. If this is False, the message is not allowed to continue through HUB_IN, and an error message is sent to the requesting system. If this flag is not specified during the update, a default of True is set. Valid values are 'True' and 'False'. UpdateMessageProfile and GetMessageProfile will operate whether this flag in their profiles is set to True or False.

WorkFlowParametersList

If Workflow processing is required, a list of Workflow parameters can be indicated. This would be fields and values that Workflow expects to receive in the initial message from MQSI.

ParameterName: Indicates the field name.

ParameterPath: Indicates where the field can be located within the message.

(**Note**: See the Dynamic Parameters Area of ProcessTemplateExecute in "Workflow Considerations" in chapter 7 for a description of the Name and Path usage.

DefaultValue: Indicates the default value of the field if it is not in the message.

RequiredParameterFlag: Indicates if the field is required. If the field is not in the message and it is required, the default value is used.

SystemInteractionList

Messages that use system interaction check must have a System Interaction List, which indicates the systems this system must communicate with. Each system within this list will receive the message.

SystemSymbolic: Indicates the symbolic of the message

MessageTypeName: Indicates the message type that must use the system. This message type should be the same as the one specified in the required fields.

RequiredInteractionFlag: Indicates if this system is required. If the system is down, no back ups are found, and the system is required, a system interaction error is generated.

SystemBackup: Specifies a back up for this system.

NOTE: The order that the SystemSymbolics in the SystemInteractionList appear in the UpdateMessageProfile message is the order in which those systems will receive the message. In a non-Workflow scenario, those systems will each be sent at the same time, each BackEnd having no concern for the other. In a Workflow scenario, each BackEnd system would receive the message serially, with the response from each one possibly affecting the message being sent to each subsequent system.

CRF

The addition of any new entries in the CRF such as parties, policies, etc., is automatic, based on the build in CRF functionality in WMQI Enabler. XML messages drive the CRF functionality, based on the information in the WMQI Enabler header. To ensure data integrity, it is recommended that entries never be manually added to the CRF database.

Disabling a message

At some point, the processing of certain types of messages may no longer be needed or desired. The following action can be taken to disable a message.

1. The MQSWF process flow can be removed.
2. The adapter(s) can be modified so that messages of certain types are no longer allowed out of or into applications (Optional).
3. The MessageType can be disabled via the UpdateMessageProfile message.

Message compression

Message compression may be desired to optimize the performance of the WMQI Enabler product, and the manner in which messages are handled. Message compression is an option within MQSeries and may be invoked, if desired.

Compression is accomplished within MQSeries by means of a Sender and Receiver exit within a specified Channel. Installation of the compression exit is very simple. It is an MQSeries offered package named MO02. The send and receive dll files are set up in the send/receive section of an MQ Channel definition.

There are 4 options:

0) No compression at all.

1) Compress the 1st segment only.

2) Compress 2nd and following segments of message if 1st message was compressed more than 10%.

3) Always try to compress multiple segmented messages.
The default is 2 (Adaptive compression).

The impact to WMQI Enabler itself is minimal since the compression is done when a message is placed on a Queue after leaving WMQI Enabler. Decompression takes place when a message is received on a Queue prior to entering WMQI Enabler.

Message persistency

The WMQI Enabler application makes extensive use of a Message_Log_Table. Each message processed though WMQI Enabler is logged to the Message_Log_Table by all Flows over which that message passes. WMQI Enabler uses its message log to provide a "chain" by which WMQI Enabler can group messages across the several Flows that message is processed. These messages are grouped together with a common process-ID that is generated by WMQI Enabler.

MQSeries includes the concept of message persistency. The decision to use message persistency has definite performance implications. Many users will choose to use non-persistent message in order to improve message throughput. WMQI Enabler respects the persistency of messages processed by it.

WMQI Enabler's message log should not be confused with MQSeries message persistency. WMQI Enabler's message logging scheme should not be viewed by users as enhancing or replacing the MQSeries message persistency.

Refer to MQSeries documentation for a complete description of message persistency.

Security and encryption support

Though data encryption and security beyond the normal bounds of the supporting products is not a part of the WMQI Enabler product currently, such encryption and security should be considered to secure the data contents of messages processed by WMQI Enabler. Different options are currently available for such data encryption and security.

SecureWay Policy Director for MQSeries, by Tivoli, supports message encryption and decryption at the queue or channel level. SecureWay Policy Director captures messages going to or coming from queues listed in the Policy Director LDAP and performs various functions including encryption.

Policy Director for MQSeries (PDMQ) is a comprehensive security solution for IBM® MQSeries. It provides access control services to restrict, which users, or applications, can, get/put messages on specific queues. It also allows MQSeries applications to send data with confidentiality and integrity using keys associated with the sending and receiving users or applications. These services are provided transparently to MQSeries since applications are supported without requiring any changes to them. PDMQ provides centralized authorization, using PKI, for strong encryption and accountability. It also provides Channel level security, through end-to-end data signing and encryption of messages.

Another approach to achieve data encryption is found in the JCE 1.2.1 (Java Cryptography Extension). The JCE Extension is packaged and installed much like the JDK1.2.2. It is packaged as an .exe file. Once installed there are configuration files within the product to be updated based on the installation choice you make. Sun offers documentation on installation and configuration of the product. An application program makes use of the classes within the JCE. This extension allows the application to encrypt and decrypt messages using public and private keys. If a channel is being used, a Send or Receive exit could be used to execute the function. If no channel is available, a process tied to a queue or a queue listener program can be employed.

Listed below is a Web site that explains what the JCE is, along with some installation and implementation tips and examples.

http://java.sun.com/products/jce/

Chapter 5
WMQI Enabler and MQSeries Integrator

Modifications to MQSeries Integrator

The MQSeries Integrator (MQSI) flows have been constructed to support the numerous features found within the WMQI Enabler product regardless of the specific XML message vocabulary used. Changes to these flows contain the potential of affecting the overall product functionality and quality. While these changes may certainly be made, it is imperative that such changes be tracked with a sound Change Control Process so acceptance of future releases would be possible without undue effort.

Since MQSI does not use a DTD to process or validate XML messages, any new DTD that is created to present command content will not require any changes to the MQSI flows. If changes are made to the message header, above the command section, these changes will also have to be reflected in the MQSI flows.

WMQI Enabler internal message flows

MQSI uses nine primary process flows to manage the information that is going in and out of WMQI Enabler. These primary flows encompass subflows implemented to provide the product features. Some of the subflows are being used in multiple flows. In addition, this chapter describes the primary flows to aid in the understanding of the WMQI Enabler product and how it functions internally.

The primary flows and their high level subflows are listed below.

HUB_IN_Flow*

- AdvancedInput_Subflow
- GetMessageProfile_Subflow
- ProcessSession_Subflow
- HubOnlyMessageRouter_Subflow
- ProcessSystemInteraction_Subflow
- ProcessWorkflowStart_Subflow
- CRF_Subflow
- SDR_Subflow
- AdvancedOutput_Subflow
- KillProcess_Subflow

- LogError_Subflow

MQWF_OUT_Flow*

- AdvancedInput_Subflow
- ProcessWorkflowRequest_Subflow
- CRF_Subflow
- LogError_Subflow
- SDR_Subflow
- AdvancedOutput_Subflow

HUB_RWF_IN_Flow*

- AdvancedInput_Subflow
- ProcessWorkflowResponse_Subflow
- SDR_Subflow
- LogError_Subflow
- AdvancedOutput_Subflow

HUB_R_IN_Flow*

- AdvancedInput_Subflow
- LogError_Subflow
- CRF_Subflow
- SDR_Subflow
- AdvancedOutput_Subflow
- KillProcess_Subflow

MQWF_END_Flow*

- AdvancedInput_Subflow
- LogError_Subflow
- LogMessage_Subflow
- KillProcess_Subflow

HUB_ONLY_ONLINE_Flow*

- AdvancedInput_Subflow
- CheckOnlineCommandType_Subflow
- LogonAndRespond_Subflow
- LogoffandRespond_Subflow
- CRF_Subflow
- SystemShutdown_Subflow
- SystemRestart_Subflow

SetSubscription_Subflow
FormatHubResponse_Subflow
SDR_Subflow
LogError_Subflow
AdvancedOutput_Subflow
KillProcess

HUB_ONLY_OFFLINE_Flow*
AdvancedInput_Subflow
CheckHubOffileCommandType_Subflow
UpdateSystemProfile_Subflow
GetSystemProfile_Subflow
GetSDREntry_Subflow
UpdateSDREntry_Subflow
KillSession_Subflow
FormatHubResponse_Subflow
KillProcess_Subflow
GetINSEntry_Subflow
UpdateINSEntry_Subflow
GetNLSEntry_Subflow
UpdateNLSEntry_Subflow
GetMessageProfile_Subflow
UpdateMessageProfile_Subflow
LogError_Subflow
SDR_Subflow
AdvancedOutput_Subflow
KillProcess_Subflow

LOG_ERROR_BACKUP_Flow*
LogOriginalMessage_Subflow

DEFAULT_ACTIVITY_FLOW*

* These Message Flows are displayed in this chapter. To view any of the other Message Flows, you may import them into the MQSI Environment on your system.

NOTE *Appendix B contains brief descriptions of every subflow.*

HUB_IN_Flow

The HUB_IN_Flow first logs messages to:

- Aid in tracking errors.
- Reduce the size of the message sent into the MQSeries Workflow product.
- Allow MQSeries Workflow to specifically send a particular message.

MQSeries Workflow can request a particular message by name as a result of this logging process. With the fields found in the MessageLog database, this logging will allow multiple message access by MQSeries Workflow of the message cache.

The Workflow_Parameters_Table provides for dynamic building of the MQSeries Workflow container section. This table allows the fields passed between MQSI and MQSeries Workflow to be determined dynamically. In addition, MQSeries Workflow processes can be made to dynamically alter the XML message being sent to the back-end. Implemented in this manner, MQSeries Workflow can perform additional tasks such as dynamic routing.

Session identification and message sequence validation are handled in the HUB_IN_Flow as dictated by the message profile. The HUB_IN_ Flow uses the Message Profile to determine whether the message should be routed through MQSeries Workflow or directly to the back-end system. In the process of building MQSeries Workflow messages, this flow also populates the ReplyTo queue and queue manager with values that govern MQSeries Workflow processing in the response messages. This ReplyTo information is defaulted to have Workflow processes respond to WMQI Enabler.

If system interaction processing is chosen for a particular message, the destination system(s) is checked to determine if the message can be delivered immediately. If not, the message will either be stored and resent when the destination system becomes active, or discarded per the store forward flag of the requesting system.

The figure below shows a graphical representation of the HUB_IN_Flow:

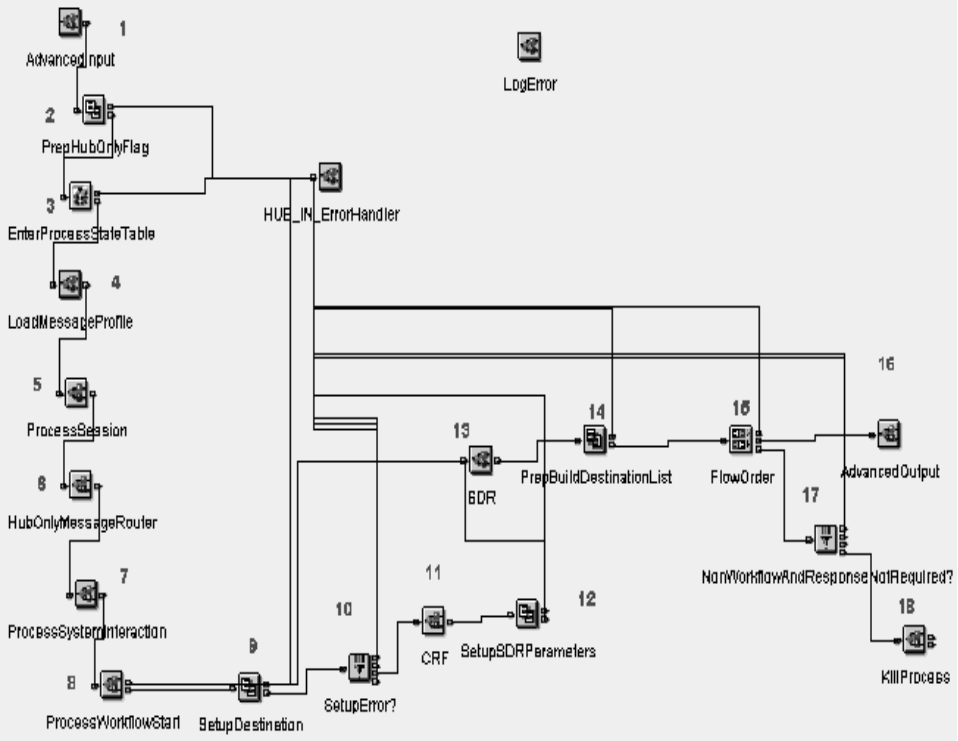


Figure 6: HUB_IN_Flow.

Verbal explanation

The HUB_IN_Flow is the workhorse of the WMQI Enabler product. All transactions will start through this flow from the queue named HUB_IN. Upon entry into the WMQI Enabler product (step 1), the complete message is placed in the MQSI_WorkArea that is used to support internal processing through WMQI Enabler. The MsgType field of the MQMD header is examined. If it is MQMT_REQUEST, a field is populated in the work area indicating that this message will need a response. Otherwise it is populated indicating that this is a MQMT_DATAGRAM, and an asynchronous request. A ProcessId is assigned to the message. If the message contains a correlation id in its MQMD header, indicating it is a response to another message, it is used to search the message log to find the corresponding process id. The message is then logged in a database.

Step 2 checks the message to determine whether it is a Hub only message or not. A flag is set within the work area to aid in future processing. The flag values are as follows: 0 = Non Hub Only, 1 = Hub Only Online and 2 = Hub Only Offline.

All messages, workflow or not, require the usage of the process id that was generated in step1. In Step 3, the ProcessId is stored in the FSE_SESS database.

With step 4, the MessageProfile is selected from the FSE_MSGP database and stored in the MQSI_WorkArea. The "bodyCategory" attribute of the message's CrfActionGroup is used to determine the message type and the profile selection. If this is a Hub Only message, the "cmdType" attribute of the COMMAND section is used to determine the profile selection.

With step 5, if the message profile indicates session processing, the session id of the message is verified as being valid. A valid session id is created when a Hub Only Online Logon command is issued. To pass this validation, the message must contain a non-timed out session id that was issued by a previous logon command. If the message profile indicates sequence processing, a comparison is then made between the message's dependency and the last message type that completed with the same session id. The message type that this message is dependent on must be the most recently completed process. Sequence Validation can only be performed if Session Validation was performed also.

Step 6 checks the message to determine whether it is a Hub only. If the message type flag created in step 1 is a '1' or '2', the message is sent to the appropriate Hub Only queue being HUB_ONLY_ONLINE or HUB_ONLY_OFFLINE.

Step 7 again uses the message profile. If the message profile indicates system interaction processing, WMQI Enabler checks the system database to see if the destinations indicated in the message profile System Interaction List are currently active. If one is not, the same check is performed on the destination system back up system, if any, as indicated in the system database. The check continues until an active system is found, or no more backups are indicated. If any of the required systems are down and do not have an available backup, the message has failed system interaction validation. If the requesting system’s profile indicates StoreMessage, the message is stored to a database to be processed when the destination system becomes active again.

In step 8, if the message profile indicates that WorkFlow processing is needed, the WorkFlow Data Container section is built, and the message bypasses the Cross Reference Function (CRF) to be routed to WorkFlow.

Step 9 determines the destination system(s) for the message. If SystemInteractionCheck was performed, the systems that were specified in the system interaction list are used. If SystemInteractionCheck was not performed, the destinationLogicalId of the message is checked. If it exists, it will be the destination of the message. If it does not exist, the DefaultDestinationSymbolic in the message

profile is used as the destination, and is placed in the destinationLogicalId of the message. If neither the destinationLogicalId or the DefaultDestinationSymbolic exist, an error is created.

Step 10 checks the error flag generated in the previous step. If it exists, the message is routed to error processing.

Step 11 performs the CRF function for those messages not requiring MQSeries Workflow. For MQSeries Workflow messages, CRF would be performed in the MQWF_OUT_Flow. CRF handles the translation of keys between the Source and Destination Systems. It also allows for the addition, deletion, or modification of keys with the WMQI Enabler product's CRF database.

Step 12 places the message destinationLogicalId on the Symbolic Destination Resolution (SDR) execution list within the MQSI WorkArea. The destinationLogicalId is equipped with the destination resolved in Step 9. If SystemInteractionCheck was performed, the systems that were specified in the system interaction list are used instead of the destinationLogicalId.

In step 13, the SDR takes all of the system symbolics on its execution list, and selects from the SDR database the queues and queue managers associated with those system symbolics. These results are stored in the SDR results list within the MQSI WorkArea.

Step 14 moves the queues and queue managers stored on the SDR results list to the BuildDestinationList's execution list within the MQSI WorkArea.

Step 15 is a flow order node that indicates the order of processing. In this case, once step 16 has completed successfully, step 17 will be processed.

In step 16, the message is logged in a database. The queues and queue managers stored in the BuildDestinationList's execution list are moved to the internal MQSI destination list structure that is used to route the message. The message is extracted from the WorkArea structure. The message is output to the queue(s) as specified in the destination list.

Step 17 checks to see if this message was an asynchronous request not requiring WorkFlow processing. If both of these conditions are true, the process id generated for this request is no longer relevant, and step 18, KillProcess, is used.

If the message fails the validation for message profile, session validation, sequence validation, or system interaction, or a system error occurs at any point, the ErrorHandler subflow is invoked. In this case an error message is built by using NLS error handling processing, deriving the error message from information provided by the system and pre-built error codes in the database. The destination of the message is changed to the "reply to" information stored in the MQMD header. If that "reply to" information exists, the hub queue manager and use hub queue manager as reply flag are checked in the message profile. If the flag is 'True' and the HubQueueManager exists, then it is used as the destination queue manager of the response. If that "reply to" information does not exist, the

destination of the message is made the same as the source id. This handling effectively puts a "return to sender" stamp on the message. The message goes through a Kill Process routine. This termination activity updates the process database, indicating the error. It also sets the active processes flag of the session as false if there are no other processes using that session.

This is the main input message flow for the WMQI Enabler Product. There are not many instances where this flow would need to be modified. The following are possibilities for modifications:

- Modifications to the XML command section that require changes to the Message Header.
- Modifications to the error logging.
- Modifications to the way the message is stored in the database.

MQWF_OUT_Flow

The MQWF_OUT_Flow, as with all flows, first logs the message. This flow makes changes to the outgoing message as specified by MQSeries Workflow. The flow stores information that MQSeries Workflow has added, such as a specified name of the response message and fields to return as parameters in the response message.

The figure below shows a graphical representation of the MQWF_OUT_Flow:

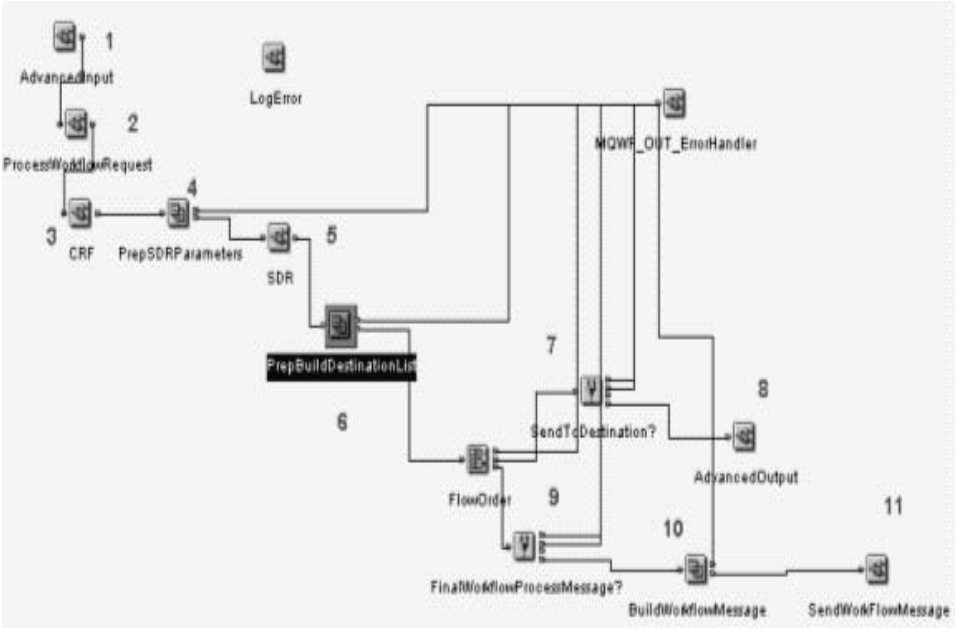


Figure 7: MQWF_OUT_Flow.

Verbal explanation

When MQSeries Workflow is either finished processing a message or is calling a target application for a program activity, the message is sent to the MQWF_OUT_Flow on the MQWF_OUT queue. This flow retrieves the original message from a database, makes modifications to that message as indicated by the process in MQSWorkFlow, and sends the message to its intended destination. Upon entry to the flow (step 1), the message is placed in the MQSI WorkArea. The MQSI WorkArea is used to support internal processing through WMQI Enabler. A ProcessId is assigned to the message. When a message is brought into MQWF_OUT_Flow, it is given the same process id that was given to the original message when it entered HUB_IN. The message is then logged in a database.

Step 2 stores the MQSFWorkFlow parameters of the message to a database. These parameters will be used again in another flow (HUB_RWF_IN_Flow), where the response message to MQSWorkflow is built. The original message is retrieved from a database and re-parsed into XML format. Changes that were indicated in MQSIWorkFlow are made to the message.

Step 3 handles the translation of keys between the Source and Destination Systems with the CRF. It also allows for the addition, deletion, or modification of keys with the WMQI Enabler CRF database. If this message did not require WorkFlow processing, the CRF functions would have been performed in the HUB_IN_Flow.

In Step 4, the SDR execution list is populated with the information of the initiator of the transaction. That information was stored in the HUB_IN_Flow when the process id record was written. If that "reply to" information exists, the hub queue manager and use hub queue manager as reply flag are checked in the message profile. If the flag is 'True' and the HubQueueManager exists, then it is used as the destination queue manager of the response. If the initiator information is not present, the destinationLogicalId of the message is used.

In step 5, the SDR takes all of the system symbolics on its execution list, and selects from the SDR database the queues and queue managers associated with those system symbolics. These results are stored in the SDR results list within the MQSI WorkArea.

Step 6 uses the workflow ProcessReplyFlag to determine if the current message is the last message the hub will process on the current process. If the flag is set to "true", the messageId that started the process is moved to the correlId of the reply message and the destinationId is set to the application that started the process. If the flag is not "true", no changes are made to the correlId or the destinationId. This step also moves the queues and queue managers stored on the SDR results list to the BuildDestinationList's execution list within the MQSI WorkArea. Also, if a success code exists in the system profile for the destination system of this response, then it is populated in the MQMD Feedback field. If the usage of the failure code is implemented, this is where modifications would need to be made.

Step 7 determines if the flow is required to send the message to another system. A Workflow may require that activity has an interaction with MQSI, but not send the message to another destination. For example, a Workflow process that requires publishing will not need a message sent to another destination. This is also used in asynchronous scenarios where the front end does not require a response. It will need the process to end so that the flow returns control to MQWF_END, where publishing occurs.

Step 8 logs the message in a database. The queues and queue managers stored in the BuildDestinationList's execution list are moved to the internal MQSI destination list structure that is used to route the message. The message is extracted from the MQSI WorkArea structure. The message is output to the queue(s) as specified in the destination list.

Step 9 determines if the message was sent from the last Workflow process activity by checking the ProcessReplyFlag in the container. This indicates that the synchronous activity requires a message from MQSI indicating everything worked correctly. The Workflow process will not complete without a response.

Step 10 creates a minimal Workflow message to be sent in response to Workflow's request.

Step 11 logs the message to the database and sends it to Workflow.

HUB_RWF_IN_Flow

This MQSI flows main function is to build a response message for MQSeries Workflow. This message is logged just prior to returning to the MQSeries Workflow system. It uses the FSE_WFCO database to return any requested fields to MQSeries Workflow in the container section of the MQSeries Workflow message header.

See the figure below for a graphical representation of the HUB_RWF_IN_Flow:

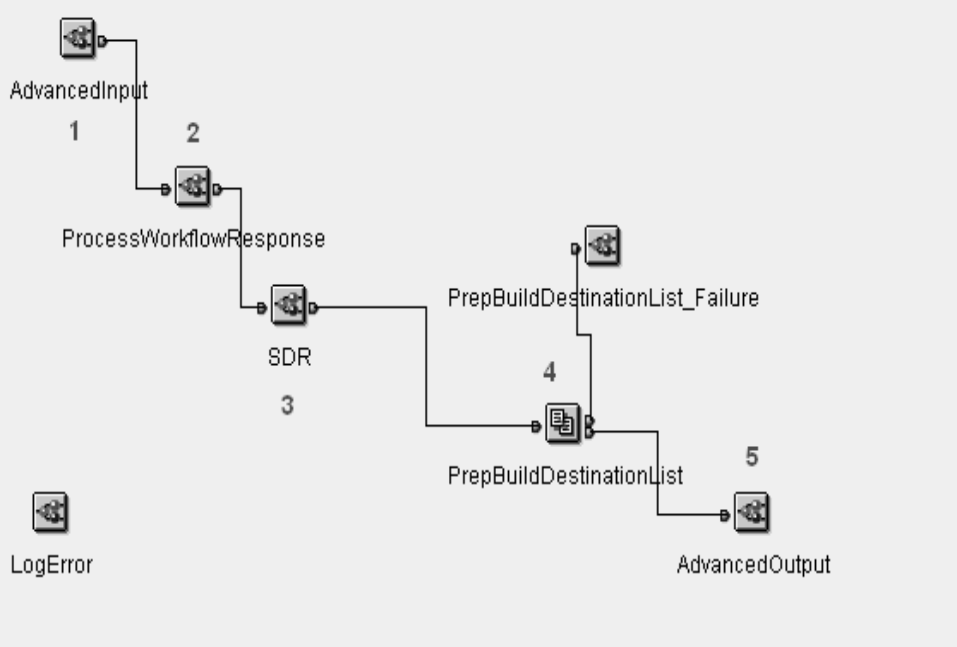


Figure 8: HUB_RWF_IN_Flow.

Verbal explanation

Once a message has gone through the WorkFlow processing and the MQWF_OUT_Flow has sent the message to the destination, the system sends its response to HUB_RWF_IN_Flow, on the queue HUB_RWF_IN. Upon entry to the flow (step 1), the message is placed in the MQSI WorkArea. The MQSI WorkArea is used to support internal processing through WMQI Enabler. A ProcessId is

assigned to the message. When a message is brought into HUB_RWF_IN_Flow, it is given the same process id that was given to the original message when it entered HUB_IN. The message is then logged in a database.

Step 2 retrieves the message profile from the database within ProcessWorkflowResponse. The WorkFlow parameters stored as part of the profile are not retrieved. Instead, those parameters that were stored in the MQWF_OUT_Flow are used. The WorkFlow message is built using those parameters. If SystemInteraction is required by the message profile, WorkFlow is expecting active flags for those systems. Since HUB_RWF_IN does not do system interaction validation, the systems are expected to be up, and the active flags are set to true. The message log database is updated with the message name, should one have been assigned by WorkFlow. The SDR execution list is populated with the WorkFlow symbolic.

In step 3, the SDR takes all of the system symbolics on its execution list, and selects from the SDR database the queues and queue managers associated with those system symbolics. These results are stored in the SDR results list within the MQSI WorkArea.

Step 4 moves the queues and queue managers stored on the SDR results list to the BuildDestinationList's execution list within the MQSI WorkArea.

Step 5 logs the message in a database. The queues and queue managers stored in the BuildDestinationList's execution list are moved to the internal MQSI destination list structure that is used to route the message. The message is extracted from the MQSI WorkArea structure. The message is output to the queue(s) as specified in the destination list.

HUB_R_IN_Flow

This MQSI flow is used for response messages from a back-end system which is not using MQSeries Workflow management for that particular message type. HUB_R_IN_Flow simply handles these responses by doing any required Cross Reference Function (CRF) and dynamically routing the message to the original sender. If these fields are empty, Symbolic Destination Resolution (SDR) will be performed on the sourceLogicalId stored from the original message in order to get the response back to the front-end system, which sent the original request message.

See the figure below for a graphical representation of the HUB_R_IN_Flow:

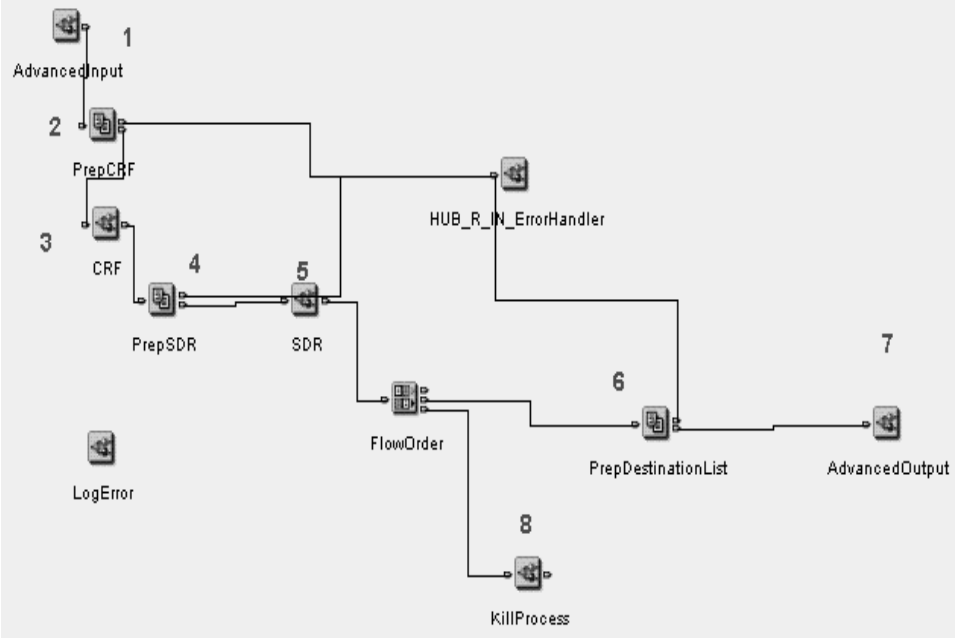


Figure 9: HUB_R_IN_Flow.

Verbal explanation

HUB_R_IN_Flow is the response flow of the WMQI Enabler product. The responses to transactions that initially come through the HUB_IN_Flow, enter WMQI Enabler in HUB_R_IN_Flow on queue HUB_R_IN. Upon entry to this flow (step 1), the complete message is placed in the MQSI_WorkArea. A ProcessId is assigned to the message. If the message contains a correlation id in its MQMD header, it is used to search the message log to find the corresponding process id. The message is then logged in a database. Generally, all messages coming into HUB_R_IN have a correlation id corresponding to the original request's message id. Therefore, the response message is assigned the same process id that was given to the request.

Step 2 populates the CRF offset in the MQSI WorkArea. The CRF function uses that offset for access to the message that contains the CRF instructions.

Step 3 performs the CRF function. Generally, the CRF command provided in the request is to reference keys. The response coming through the HUB_R_IN_Flow usually indicates instructions to add, modify, or delete keys.

In Step 4, the SDR execution list is populated with the information of the initiator of the transaction. That information was stored in the HUB_IN_Flow when the process id record was written. If that "reply to" information exists, the hub queue manager and use hub queue manager as reply flag are checked in the message

profile. If the flag is 'True' and the HubQueueManager exists, then it is used as the destination queue manager of the response. If the initiator information is not present, the destinationLogicalId of the response message is used.

In step 5, SDR takes all of the system symbolics on its execution list, and selects from the SDR database the queues and queue managers associated with those symbolics. These results are stored in the SDR results list within the MQSI WorkArea.

Step 6 moves the queues and queue managers stored on the SDR results list to the BuildDestinationList's execution list within the MQSI WorkArea. Also, if a success code exists in the system profile for the destination system of this response, then it is populated in the MQMD Feedback field. If the usage of the failure code is implemented, this is where modifications would need to be made.

Step 7 logs the message in a database. The queues and queue managers stored in the BuildDestinationList's execution list are moved to the internal MQSI destination list structure that is used to route the message. The message is extracted from the MQSI WorkArea structure. The message is output to the queue(s) as specified in the destination list.

Upon successful completion of step 7, the message is sent to the KillProcess Subflow.

In step 8, the KillProcessSubflow updates the process database, indicating the process is complete. It also sets the active processes flag of the session as false if there are no other processes using that session.

MQWF_END_Flow

This MQSI flow is exercised to end an MQSeries Workflow process. The reply message returning to MQSeries Workflow can tell the system that it is now "okay" for that system to go down. This system control element is derived from the system profile.

The flow takes place in three steps:

- First, the message is logged;
- The associated process is terminated; and,
- The MQSeries Workflow message is logged again to evidence that the process has been terminated.

See the figure below for a graphical representation of the MQWF_END_Flow:

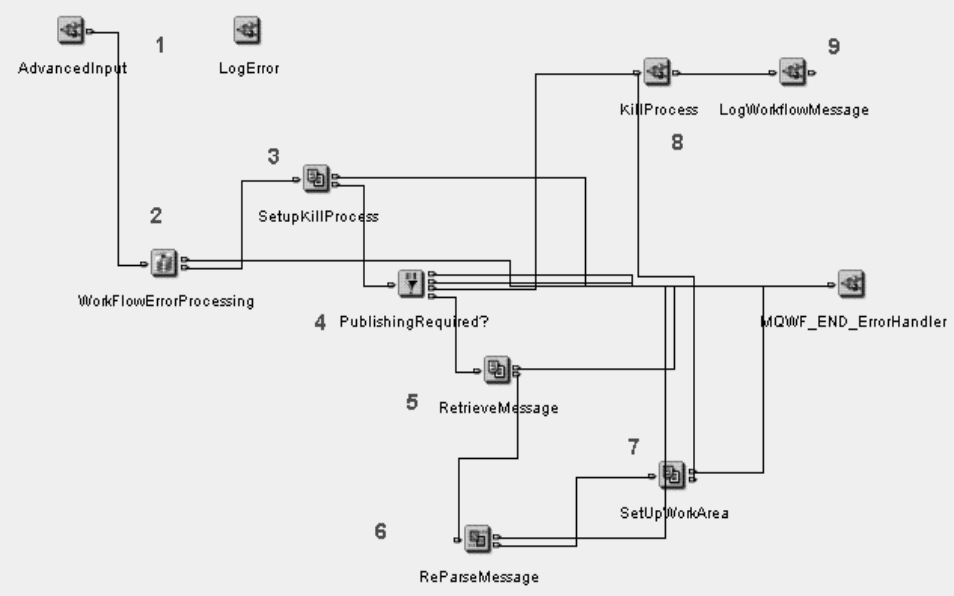


Figure 10: MQWF_END_Flow.

Verbal explanation

When MQSeries Workflow is finished with processing a message, a message is sent to MQWF_END_Flow on the MQWF_END queue upon completion of routing to MQWF_OUT_Flow. This processing is done serially. When the MQWF_OUT_Flow sends the original message to the destination, the MQWF_END_Flow terminates the process associated with the message. Upon entry to the hub (step 1), the message is placed in the MQSI WorkArea. The MQSI WorkArea is used to support internal processing through WMQI Enabler. A ProcessId is assigned to the message. When a message is brought into MQWF_END_Flow, it is given the same process id that was given to the original message when it entered HUB_IN. The message is then logged in a database.

Step 2, logs an error or exception to the error table if Workflow indicated that there was one.

Step 3 prepares the message for Kill Process. It places the Process Id and Session Id in the location within the message that Kill Process is expecting it to be.

Step 4 determines if Workflow requires a message to be published.

Step 5 retrieves, from the database, the message that Workflow designated should be published.

Step 6 re-parses the message retrieved from the database into XML format.

Step 7 places the re-parsed message into a message item in the work area. It also sets up the required publishing information.

In step 8, the KillProcessSubflow updates the process database, indicating the process is complete. It also sets the active processes flag of the session as false if there are no other processes using that session. If publishing is required, it is done so in KillProcess.

Step 9 logs the message to a database.

HubOnly flows

This MQSI flow is actually divided into two other MQSI flows. These two flows are the HUB_ONLY_ONLINE_Flow which will allow the updating of WMQI Enabler hub database tables while the system is active; the other flow is HUB_ONLY_OFFLINE_Flow which is employed for viewing and updating the hub database tables. The standard messages used in these two MQSI flows are:

HUB_ONLY_ONLINE_Flow

The HubLogonRequest provides an authentication facility for the users of the WMQI Enabler product. The flow sends a message to the queue to have authentication identification validated and a session identification assigned. The HubLogoffRequest then releases this session identification.

See the figure below for a graphical representation of the HUB_ONLY_ONLINE _Flow:

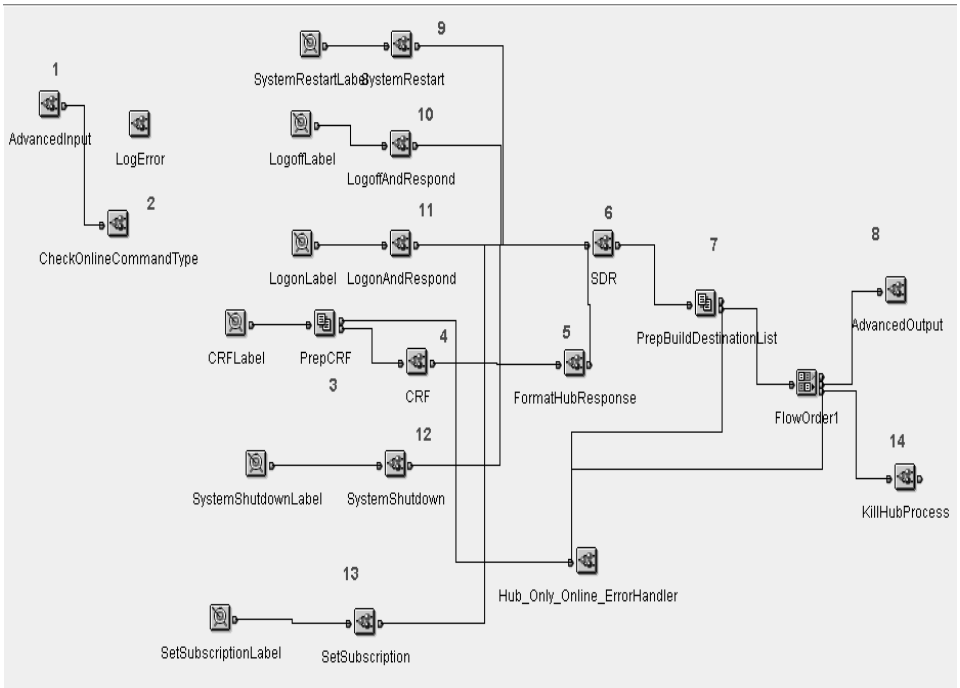


Figure 11: HUB_ONLY_ONLINE_Flow.

Verbal explanation

The HUB_ONLY_ONLINE_Flow handles online commands to the WMQI Enabler hub. These commands are SystemRestart, LogoffAndRespond, LogonAndRespond, SetSubscription, CRF, and SystemShutdown. The HUB_ONLY_ONLINE_Flow retrieves messages from the HUB_ONLY_ONLINE queue.

Step 1 places the message in the MQSI WorkArea. The same process id that was assigned to this message when it entered Hub In is reassigned to the message.

In step 2, the cmdType attribute of the Command section of the message is examined to determine the type of request being issued.

Step 3 prepares the message if this is a CRF command. The CRF offset is set.

In step 4, the message for CRF processing contains a CrfActionGroup identical to the one found in any other message. Generally, this CrfActionGroup would be used for CRF database maintenance of adding, deleting or modifying keys.

Step 5 builds the response message. The message type is set to Reply. The "reply to" information in the MQMD header is used to populate the BuildDestinationList execution list. If the "reply to" information does not exist, the sourceLogicalId in the message header is used as the destination of this message. The other commands in HUB_ONLY_ONLINE_Flow build their response messages within their sub-flows.

In step 6, the SDR takes all of the system symbolics on its execution list, and selects from the SDR database the queues and queue managers associated with those system symbolics. These results are stored in the SDR results list within the MQSI WorkArea.

Step 7 moves the queues and queue managers stored on the SDR results list to the BuildDestinationList's execution list within the MQSI WorkArea.

Step 8 logs the message in a database. The queues and queue managers stored in the BuildDestinationList's execution list are moved to the internal MQSI destination list structure that is used to route the message. The message is extracted from the MQSI WorkArea structure. The message is output to the queue(s) as specified in the destination list.

Step 9 first sets the system as being active in the database for the SystemRestart command. It then checks to see if any messages were stored for that system while it was down. These messages are retrieved from the database, and sent to the HUB_IN_Flow for reprocessing. The message returned contains a results message indicating success and the number of messages that were resent.

Step 10 checks to see if the session requesting logoff has active processes for the Logoff command. If it does not, the session is deactivated in its database. If it does have active processes, the logoff is denied. The response indicates success or failure.

Step 11 checks the authenticationId within the message header to see if it is in the System_Authentication_Table for the Logon command. If it is, a session id is issued and returned in the response. If the message does not contain a valid authentication id, the logon is denied.

Step 12 updates the databases to indicate the system's desire to shutdown for the SystemShutdown command. It checks for processes currently using the system. When SystemInteractionValidation in the HUB_IN_Flow checks system status, it does a comparison on the requested shutdown flag and the active processes flag. For this reason, once the processes on the system are finished, system shutdown takes effect.

Step 13 allows a system to add and delete subscriptions for various topics. A topic is the same as a message type. When a message reaches a publishing point, (located in KillProcess), all the systems subscribed to that topic receive the message.

Step 14 processes once the message is successfully sent through AdvancedOutput. The ProcessId assigned the Hub Only command as it entered Hub In is killed here.

WMQI Enabler commands

In the case of HUB_ONLY_ONLINE_Flow the following XML messages are targeted towards the hub as hub commands on an active system:

LogonAndRespond
This command allows the utilization of session management giving the system an opportunity to log on to a session. A session id will be returned to the system and should be included in all future messages in the same session.

LogoffAndRespond
This command logs a system off from a session. It can fail if there are still active processes running.

SystemShutdown
This command handles messages sent into the HUB by a system requesting a shutdown. A response will be formed telling the requesting system if the request was successful or not. If the response was successful the system will be allowed to shutdown. Otherwise, if processes are using the system or are in system interaction check, the system will have to wait until all of those messages are cleared from the pipeline to go down.

SystemRestart
This command notifies the HUB that a particular system is back up and running. It lets the HUB know it can send messages to it.

CRF
This command provides for the management of keys represented in the XML header form to an internal object form (ConvertToCRFObjectFormat).

SetSubscription
This command allows the addition and deletion of subscriptions for a particular system. The SetSubscription message will specify a topic and queue and queue manager to which on-topic messages will be published.

Tester
As an example, these messages can be found under the WMQI Enabler Tester directory.

HUB_ONLY_OFFLINE_Flow

This main flow handles HUB_ONLY maintenance messages. It is capable of updating system profile, message profile and SDR tables. Also, it can send back copies of records from any of these tables. Lastly, it can kill a session or a process as needed by an outside system.

See the figure below for a graphical representation of the HUB_ONLY_OFFLINE _Flow:

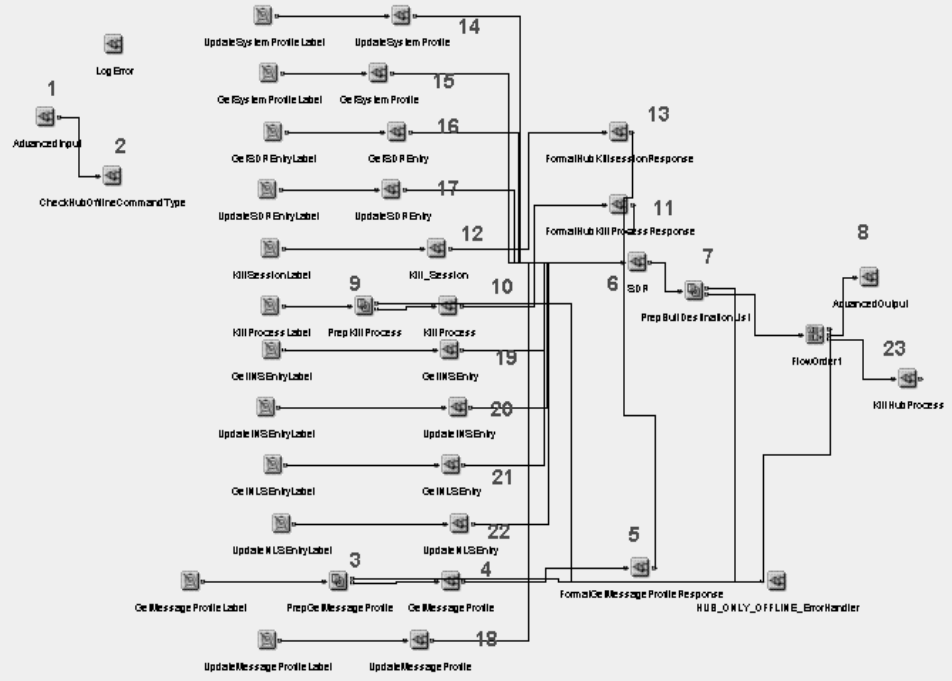


Figure 12: HUB_ONLY_OFFLINE_Flow.*

*The HubRequest messages are used for internal hub database management and maintenance.

Verbal explanation

The HUB_ONLY_OFFLINE_Flow handles offline commands to the WMQI Enabler. These commands are UpdateSystemProfile, GetSystemProfile, GetINSEntry, UpdateINSEntry, GetNLSEntry, UpdateNLSEntry, GetMessageProfile, and UpdateMessageProfile. The HUB_ONLY_OFFLINE_Flow retrieves messages from the HUB_ONLY_OFFLINE queue.

Step 1 places the message in the MQSI WorkArea. The process id assigned to the message when it first entered Hub In is reassigned to the message.

In step 2, the cmdType attribute of the Command section of the message is examined to determine the type of request being issued.

Step 3 prepares a new message within the work area for the GetMessageProfile command. This message will be used as the response that will take the message profile back to the requester.

In step 4, the GetMessageProfile Subflow retrieves the profile from the database for the message type in the CrfActionGroup or cmdType in the COMMAND section of the requesting message.

Step 5 copies the retrieved profile into the Command section of the response message. The response message is formatted to return to the message sender. If the MQMD "reply to" information is present, it is used as the destination of the response. If not, it uses the original message's sourceLogicalId.

In step 6, the SDR takes all of the system symbolics on its execution list, and selects from the SDR database the queues and queue managers associated with those system symbolics. These results are stored in the SDR results list within the MQSI WorkArea.

Step 7 moves the queues and queue managers stored on the SDR results list to the BuildDestinationList's execution list within the MQSI WorkArea.

Step 8 logs the message in a database. The queues and queue managers stored in the BuildDestinationList's execution list are moved to the internal MQSI destination list structure that is used to route the message. The message is extracted from the MQSI WorkArea structure. The message is output to the queue(s) as specified in the destination list.

Step 9 prepares the message for the KillProcess command. The ProcessId field in the MQSI WorkArea is set from the process id sent in the command area of the message.

In step 10 the KillProcessSubflow updates the process database, indicating the process is complete. It also sets the active processes flag of the session as false if there are no other processes using that session.

Step 11 formats the response message to return to the message sender. If the MQMD "reply to" information is present, it is used as the destination of the response. If not, it uses the original message's sourceLogicalId.

Step 12 represents the KillSession Subflow. As it is implied by it's name, this sub flows kills the session that is supplied to it. Along with invalidation this session, it kills the processes that are currently using it.

Step 13 formats the response message to return to the message sender. If the MQMD "reply to" information is present, the hub queue manager and use hub queue manager as reply flag are checked in the message profile. If the flag is 'True' and the HubQueueManager exists, then it is used as the destination queue

manager of the response. Otherwise, the reply to information is used as the destination of the response. If the reply to information is not present, the original message's sourceLogicalId is used.

In step 14, the UpdateSystemProfile command updates the system profile database with the information contained within the command section of the message. This update includes the system status, the system's back up list, and the store forward parameters for that system. The records that currently exist for that system are deactivated by setting the DATE_TIME_OFF field for those records to the current time stamp. New records are added to the database with the new profile information. If the update command does not include back up systems, for example, there will be no back up systems recorded for that system. The response message is formatted within the UpdateSystemProfile Subflow.

Step 15 returns the system information for the system symbolic in the command section of the GetSystemProfile message. The information returned includes the system back up list and store forward parameters. The response message is formatted within the GetSystemProfile Subflow.

Step 16 updates the SDR information for a specific system symbolic.

Step 17 returns the SDR information for a specific system symbolic.

In step 18, the UpdateMessageProfile command deactivates the current records for the message type in the Command section, and creates new records within the databases. The information updated includes the message profile, system interaction table, and the workflow parameters table. If there are no workflow parameters indicated in the message, for example, the table will be updated to not have any parameters that correspond to the message type.

Step 19 allows you to get the INS information for the current Hub set up. This information includes codes for the hardware platform, product version, and default language.

Step 20 allows you to update the INS information.

Step 21 allows you to get the information about an NLS error message based on an error number.

Step 22 allows you to update or add a new NLS error message into the database.

Step 23 kills the process id that was assigned to this message in Hub In.

WMQI Enabler commands

In the case of HUB_ONLY_OFFLINE the following XML messages are targeted towards the hub as WMQI Enabler Maintenance commands.

UpdateSystemProfile

This command adds or updates to a particular system profile record as stored in the Hub, which enables the user to update their system's profile in the WMQI Enabler databases.

GetSystemProfile must be called beforehand to get the existing values in the database. The UpdateSystemProfile always does a complete replace, so data that is not included is deleted.

GetSystemProfile

This command requests a particular system profile record as stored in the Hub, which enables the user to get a response message showing what their system's profile looks like in the WMQI Enabler databases.

GetSDREntry

This command requests a particular SDR entry as stored in the Hub, which enables the user to get a response message showing what their system's SDR entry looks like in the FSE_SDR database.

UpdateSDREntry

This command adds or updates a particular SDR record as stored in the Hub, which enables the user to update their system's SDR entry in the FSE_SDR database.

GetSDREntry must be called beforehand to get the existing values in the database. The UpdateSDREntry always does a complete replace, so data that is not included is deleted.

KillProcess

This command is sent to kill a particular process. Once it has been killed, any systems that have requested shutdown but were held up by a Process that has just been killed can be sent a message telling them it is now safe to shut down.

This condition is only true if the process being killed was the only one using the system.

The publishing function has been added here as an example. Publishing may be turned on by setting flags in the message profile with an UpdateMessageProfile message. Subscriptions can be set by sending SetSubscription messages to the HUB_ONLY_ONLINE_Flow.

KillSession

This command is sent to kill a particular session.

UpdateMessageProfile

This command adds or updates message profile record in the FSE_MSGP database. The content of this record can consist of enabling session validation, system interaction checking, workflow management, setting workflow parameters, and sequence validation for this particular message type.

GetMessageProfile must be called beforehand to get the existing values in the database. The UpdateMessageProfile always does a complete replace, so data that is not included is deleted.

GetMessageProfile

This command requests a particular message profile as stored in the Hub, which enables the user to get a response message showing what a message profile looks like for a particular message type in the FSE_MSGP database.

GetINSEntry

This command requests the active INS information from the INSTALL_DATA_TABLE in the FSE_SYSP database.

UpdateINSEntry

This command updates the INS information within the FSE_SYSP database.

GetNLSEntry

This command returns the error message information for a particular code. These codes are stored with their two digit language number concatenated with the four digit error number. For example, the error code of 1499 is stored in English as 101499.

UpdateNLSEntry

This command allows you to update or add a new error code into the NLS error message database.

As an example, these messages can be found under the MQTester directory.

MQWF_DEFAULT_ACTIVITY_Flow

This main flow provides a "default" flow that can be utilized by workflow activities. The flow simply accepts an incoming message, maps the input data to an output and returns a response containing the mapped data back to a specified queue.

See the figure below for a graphical representation of the MQWF_DEFAULT_ACTIVITY_Flow:

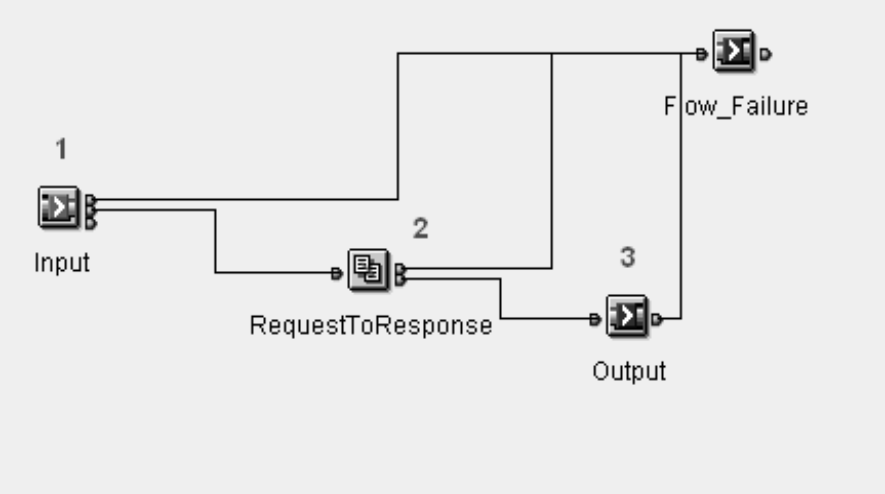


Figure 13: MQWF_DEFAULT_ACTIVITY_Flow.

Verbal explanation

Step 1 sets up the Destination List using the MQMD header information to define the response queue information.

Step 2 builds a WorkFlow response message from the WorkFlow request message.

Step 3 passes the request data, unchanged, to the response data.

Step 4 writes the response to the Queue specified in the Destination List.

StoreMessageTemplate_Flow

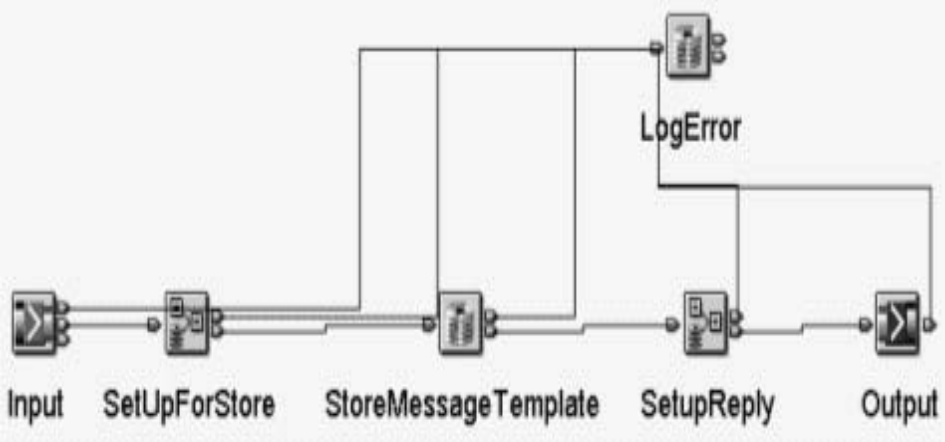


Figure 14: StoreMessageTemplate_Flow

This flow is used to store a message template in the Message_Template_Table. The Template Message has the following form:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Message SYSTEM "MQSFSE_2001.dtd">
<Message>
     <SourceQueue>FEIN</SourceQueue>
     <SourceQueueManager>MQSIQM</SourceQueueManager>
     <Action>INSERT</Action>
     <Name>AddParty</Name>
     ß---- Message Content ---à
</Message>
<!--filename=AddParty.xml-->
```

Possible actions are INSERT, UPDATE, and DELETE. The INSERT and UPDATE commands delete any existing records for the template specified in the Name tag (if any), and adds a new record with the current template for the specified name. DELETE deletes any existing records associated with the information in the Name tag. The SourceQueue and SourceQueueManager tags specify where the requesting system would like the response of the store template operation sent. The MQMD header sent in with the template is stored as part of the template.

- MQInput Node Input reads from the TEMPLATE_IN queue.

- Compute Node SetUpForStore saves the information in the declaration line to a temporary space in the destination list and nulls out the declaration line in the XML message.

  It also stores the SourceQueue, SourceQueueManager, Action, and Name information to the destination list and nulls out those tags within the message. That information is for performing this operation, and should not be stored with the template.

- Database node StoreMessageTemplate first does a check on the action specified in the message. If the action is INSERT, UPDATE, or DELETE, a delete of any existing records with the Name specified is performed.

  If the command is INSERT or UPDATE, the MQMD header and message are stored to the Message_Template_Table. The MQMD header is stored as individual columns. The information from the declaration line is stored in individual columns. The message template is bitstreamed into BLOB format. The doctype at the top of the message is stored in its own column.

- Compute node SetupReply sets up a minimal response message. If the Source tags were present in the message, that data is used to populate the MQMD reply to information. If it is not present, the existing MQMD reply to information is used.

- MQOutput node Output sends the created response message to whatever information was determined in the previous step.

- Database Node LogError is the end point of all failure paths in this flow. It stores the existing message in the Message_Table. It also stores whatever exception information was generated into the Exception_Table. This node does not generate any response messages to be sent to the requesting system.

LOG_ERROR_BACKUP_Flow

This main flow provides a backup mechanism for logging an error to the Error Table

See the figure below for a graphical representation of the LOG_ERROR_BACKUP_Flow:

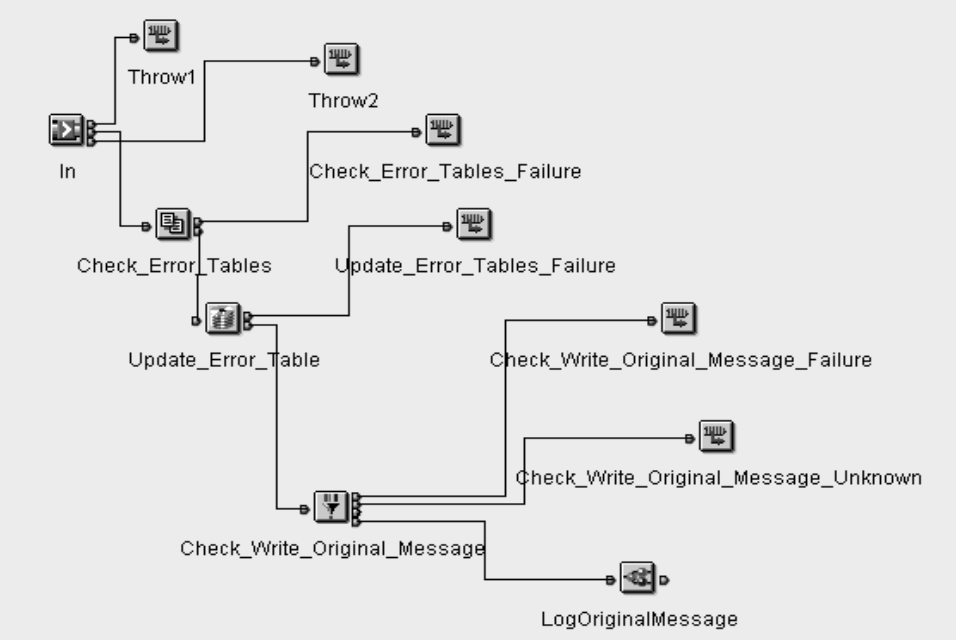


Figure 15: LOG_ERROR_BACKUP_Flow.

Verbal explanation

Step 1 checks to see if end trace, error, original message, exception messages and an error message exists for the message id.

In the event that any of the required information from step 1 is not contained in the error message, Step 2 updates the Error_Table accordingly.

Step 3 logs the original message if there is one.

LogMessage_Subflow

This subflow provides for the determination of a Process id and logs a message in the FSE_MSGL database with the correct Process id.

This subflow will either create a new process id or get a process id of a message already logged in the FSE_MSGL database that is related to the current message being processed. Alternatively, if this subflow determines it has a message that is starting some process off, and there is no other message related to it, it will generate a new process id. A process id is assigned to any message with a non-NULL msgId and is logged with that process id. If the incoming message is related to an existing message, it uses the process id previously generated.

See the figure below for a graphical representation of the LogMessage_Subflow:

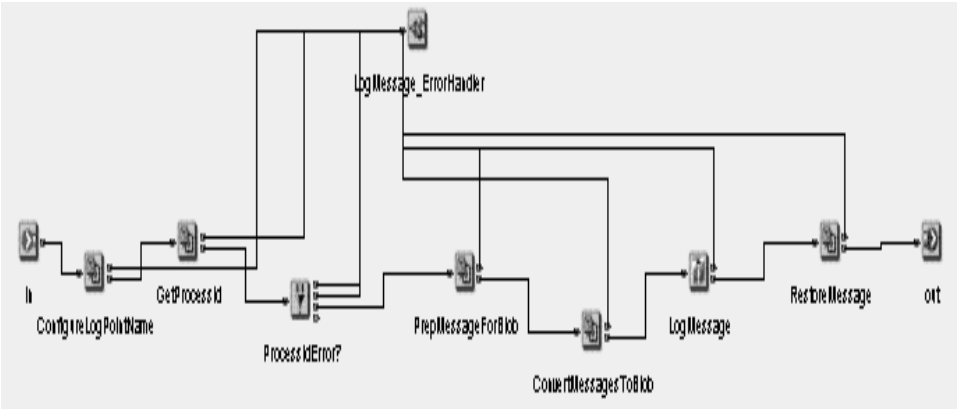


Figure 16: LogMessage_Subflow.

CRF_Subflow

The CRF_subflow handles the functionality of the Cross Reference File. This subflow controls additions. deletions, modifications and translations of key and key information held in the CRF_Table on the CRF database. The subflow processes incoming message information, by CRFActionGroup, by KeyGroup. Incoming messages content is validated against message content and also against existing CRF_Table data. If the message content is valid, the subflow processes all deletions, then all modifications, then all additions, then all translations. If the message content is invalid, or any other processing error occurs, the ErrorHandler subflow is invoked.

CRF utilizes the SDR_Subflow to validate SDR entries. If SDR_IN_LDAP is required, the CRF_Subflow should be modified to incorporate the LDAP Subflow.

See the figure below for a graphical representation of the CRF_Subflow:

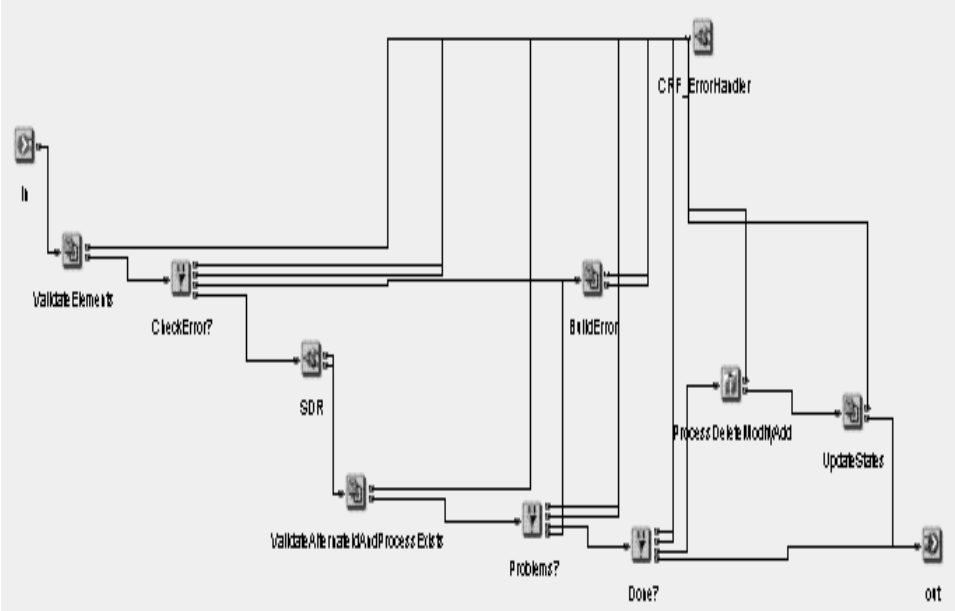


Figure 17: CRF_Subflow.

Changing the code page

WMQI Enabler assumes that messages routed through it are based on the UTF-8 code page. The value 1208 in the MQMD header field CodedCharSetId specifies UTF-8. In UTF-8, the strings **<ToBLOB>** and **</ToBLOB>** are represented as the Hex Code values **3c546f424c4f423e** and **3c2f546f424c4f423e**, respectively.

If the CodedCharSetId is changed to a value other than 1208, then the strings **<ToBLOB>** and **</ToBLOB>** may not be represented by the same Hex Code. If the Hex Code values are not exactly the same in your new code page, then you must modify code in two MQSI nodes.

The MQSFSE_LogMessage_Subflow, *ConvertMessagesToBlob* node and MQSFSE_StoreMessage_Subflow, *StoreMessage* node have the following BLOB/Hex code values in the code:

```
SET startTag ='3c546f424c4f423e'; -- <ToBLOB>
SET endTag ='3c2f546f424c4f423e'; -- </ToBLOB>
```

Both nodes must be modified such that the Hex Codes in the desired code page are used to represent the ToBLOB and /ToBLOB strings.

NOTE *If the length of the Hex Codes change, the SQL code around it must also change to reflect the difference in length.*

Chapter 6
WMQI Enabler capabilities

WMQI Enabler was designed with many capabilities in mind. Its structure was designed to have connections to multiple front-end and back-end systems. It was also designed to allow for session management and message sequence validation. The architecture of WMQI Enabler was designed for high performance under stress. The CRF functionality allows connections between disparate systems of keys attached to related entities. WMQI Enabler is capable of performing dynamic routing of messages to single or multiple destinations. Systems can choose to have messages destined for inactive systems stored and forwarded later. Finally, WMQI Enabler maintenance can be performed by sending messages which interact with the HUB only.

HUB commands

A set of HUB_ONLY_ONLINE and HUB_ONLY_OFFLINE XML message commands are supported in WMQI Enabler. The commands include requests for internal database maintenance, Logon and Logoff, System startup notify or shutdown requests, update SDR profile, get and update system profile, register and drop subscription requests, kill session requests, get or update NLS error messages, get or update install data requests, kill process requests, and get or update message profile requests.

Message routing interface

The message routing interface supports alternative message routing. The XML messages can be structured to request interface support with:

- The Hub (HUB_ONLY_ONLINE and HUB_ONLY_OFFLINE).
- The Hub and user applications.
- The Hub with MQSeries Workflow and user applications.

Sequence validation

Sequence validation support allows the ordering of messages in the same session through WMQI Enabler. Each message type can use the MessageTypeDependency flag in the Message Profile to indicate a message type

must complete successfully in the same session before the current message type can begin processing. The sequence of each message can be validated one at a time to govern a list of message types.

Care should be taken that not all messages are dependent upon the processing of another message, as some messages must always be first and not dependent upon other messages. These messages must not rely on sequence validation.

Interaction check

Interaction check support consists of the System Interaction and System Profile databases containing a list of valid front-end and back-end systems with which to interact and identifies whether these systems are available, respectively. With these indicators, WMQI Enabler can check whether active requests to a system can be made, i.e. whether a system is up, going inactive, or is inactive.

Symbolic destination resolution

The Symbolic Destination Resolution (SDR) utilizes the Symbolic Destination Resolution (FSE_SDR) database. This database contains a list of symbolic names to identify systems, the Queue Manager and queues used to communicate with them. In addition, the SDR supports a backup list for systems that can be used to send a message if the 'primary' system is unavailable (assuming, of course, the backup system can successfully receive/process the message).

Session validation

The WMQI Enabler product supports the Logon command to establish a session ID to be used on subsequent messages. Session validation checks whether the session exists and hasn't timed out. If timed out, a Logon can be requested again using the same authorization information to re-establish a session or a Logoff may be issued to kill a session.

CRF

All cross-reference functionality takes place in this subflow. This subflow can be called from a number of the main flows including:

HUB_ONLY_ONLINE_Flow

MQWF_OUT_Flow

HUB_R_IN_Flow

HUB_IN_Flow

CRF functions allow the user to add, modify, delete, reference and translate keys.

Pub/Sub

When an application responds to a request message, the response is forwarded for publication to interested subscribers. This will allow a subscriber to deactivate, and then upon re-activation solicit the hub for all subsequent changes on a topic. This allows the subscriber to re-synchronize and allows hub members to only publish deltas for particular topics. The Pub/Sub features of WMQI Enabler use the internal publish/subscribe functionality provided with MQSI.

Pub/Sub can be set up as follows:

1. Publishing can be turned on by setting flags in the message profile with an Update Message Profile message. These fields can be reviewed in the message profile requirements in Chapter 4.
2. Subscriptions can be set by sending SetSubscription messages (Hub only Online command message) to WMQI Enabler (HUB_IN_Flow). SetSubscription commands are outlined in Chapter 5. Once a subscription is successfully set, it can be viewed under the Subscriptions tab of the MQSI workspace.
3. Any topics (equivalent to MessageType) must be added and deployed under the Topics in MQSI Control Center. (More documentation on this subject is available in MQSI.) Messages can then be published if the message type matches a publishable topic.

   It is important to note that, in the current pub/sub implementation, publishing only occurs as processes end, in the KillProcess subflow. Therefore, publishing would only occur in a successful request/response model in the HUB_R_IN_Flow or in the MQWF_END_Flow or as an asynchronous message is sent from HUB_IN. That is, a message must be a response from a "back end" or a final communication from WorkFlow in order to be published. Publishing can be set to occur when an error occurs for a particular message type. In this regard, publishing can occur in all system error scenarios.
4. Systems with subscriptions to that topic will have those messages published to a queue that was set up when the subscription was registered.

PluggablePublish_Subflow

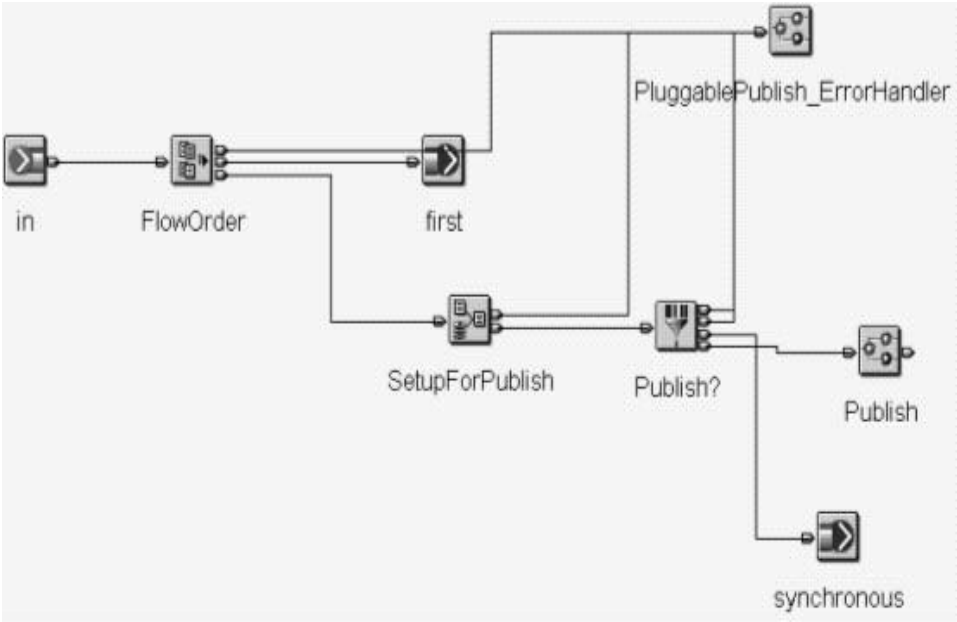


Figure 18: PluggablePublish_Subflow

The above subflow provides the ability to plug in the publish functionality anywhere in the WMQI Enabler flows that the MQSIWorkArea exists in the message. For example, this means that it must be used anywhere after the AddWorkArea_Subflow has been used and before the RemoveWorkArea_Subflow has been used.

This subflow begins with a FlowOrder node. In most scenarios which require ordered processing, whatever flow is attached to the output node "first" would finish processing successfully, and then the publish functionality is processed. The flow gets required fields from the message profile and the message, sets up the publish offset, checks the fields to see if publishing is to be performed, and then sends the message to the Publish subflow. At that point an output terminal labeled "synchronous" exists. Processing that must be done in parallel to the publishing would be attached to this terminal. For example, in an error situation, a DatabaseRollback node would be done in parallel to publish processing.

Any flows that would be processed after publishing occurs would need to be connected to the synchronous output terminal.

Optional support of LDAP

An LDAP protocol is a vehicle for accessing a directory. It defines the operations one may perform, such as search, add, delete, modify, and change name. It also defines how operations and data are conveyed. The information model and name space are based on Entries. An entry is simply a place where one stores attributes.

NOTE *LDAP support capability is provided as is and limited to Windows NT systems only.*

The LDAP standard:

- Defines a network protocol for accessing information in the directory.
- Provides an information model defining the form and character of the information.
- Provides a name space defining how information is referenced and organized.
- Both the protocol itself and the information model are extensible.

Enhanced authentication

Session validation will encompass all messages within the Hub except for the Logon and Logoff commands. Valid session id's are created and assigned through the Logon command. Because of this, it must be allowed to process without a valid session id. The Logoff command is allowed to operate on a session id that has timed out. This would fail session validation.

NLS error handling

Error Messages using the National Language Support (NLS) Standard provide developers and users a structured way to create and read errors and informational messages. Each message is identified by a 9-digit number that represents the language, system, and location that the error or informational message occurred. For the WMQI Enabler implementation, NLS standards will be applied to:

1. Uniquely identify where an error or information message occurred, i.e. Flow /subflow/node .
2. Allow for multiple languages.
3. Move away from hard coded message text.
4. Allow for cosmetic formatting of error message text.

5. Position the WMQI Enabler product for future error handling and informational messaging enhancements.

The National Language Support (NLS) process within WMQI Enabler is performed when one of the MQ products or a node within the WMQI Enabler application generates an error. A message number of 1234 can be displayed in English or French because the NLS_Error_Message_Table can contain message text in any language. The error number does not change, just the language it is to be presented in. WMQI Enabler NLS also allows you to add text or program variable data to a message. NLS error messages are held in the NLS_Error_Message_Table within the FSE_SYSP database.

Error descriptions

Reserved	0000 - 0999 <== reserved for development use
General	1000 - 1499
CRF	1500 - 1999
Reserved	9000 - 9999 <== reserved for customer use

Error Code	Description
1001	Invalid Body Category - specifies the body category which is invalid. Verify that the body category specified in either the bodyCategory of the CrfActionGroup or the cmdType in the COMMAND section of the hub only message has a message profile which exists in the Message_Profile_Table.
1002	Message Type Disabled - specifies the message type that is disabled. This message indicates that the body category specified in either the bodyCategory of the CrfActionGroup or the cmdType in the COMMAND section of the hub only message has a message profile within the Message_Profile_Table that has the MQSI_MSG_ENABLED_FLG set to False.
1003	Invalid Session ID - specifies the session id state which is either Invalid or TimedOut. The sessionId attribute in the message header is Invalid if it contains a value that is not in the Session_Table. SessionId's are issued through the Logon command. The sessionId is TimedOut if it exists on the table, but has been inactive for the time out interval specified in the Session_Table. The interval is set at 60 minutes.

1004	Invalid Sequence - contains the message type dependency for the message being processed. This indicates that the current message requires it's dependency to have been the last processed message type for the same session id.
1005	Invalid Command Type - The hub only command being processed does not have a valid cmdType in the COMMAND section of the message.
1006	Null Message Id - The message sent into the hub for processing does not have a Message Id in it's MQMD header. The attribute name in the header is MsgId.
1007	Error creating/retrieving process id - If the message sent into the hub has a CorrelId in the MQMD header, a corresponding message with a message id that is the same as the CorrelId must exist in the Message_Log_Table. These messages would be part of the same process and would use the same ProcessId. If that corresponding message does not exist, this error is thrown.
1008	Add Work Area Failure - specifies the invalid message type. If the message sent into the hub does not have a high level tag of Message or WfMessage, this error is thrown.
1009	System Interaction Problems - specifies down system. If system interaction checks are performed and find that a required system is not active, this error is generated and specifies which required systems are down.
1010	Queue not found in SDR table - specifies symbolic which failed processing. All system symbolic used as destinations and sources must have a valid entry in the SDR_Table. If the symbolic does not exist in the table, or it's entry does not specify a queue, this error is thrown.
1011	Queue Manager not found in SDR table - specifies symbolic which failed processing. All system symbolic used as destinations and sources must have a valid entry in the SDR_Table. If the symbolic's entry does not specify a queue manager, this error is thrown.

1012	The following required information is null - specifies which required information is NULL. When a message is sent from Workflow into MQWF_OUT, a check is made on information that is required from Workflow for proper MQSI processing. If one of the items tha is required does not exist, this error is thrown.
1013	DefaultDestination not provided in profile. Unable to determine destination for message type: For messages that do not go through workflow or use system interaction check, the destinationLogicalId of the message is used to determine the destination. If the destinationLogicalId does not exist, the DefaultDestinationSymbolic in the message profile is used. If that does not exist, the destination is not able to be determined.
1499	System Generated Exception - This exception is genererated when the MQSI product triggers an error, i.e. database exceptions, parsing errors etc. Error 1499 will also be used if a user generated exception is received in the NLS Process Error subflow and the NLS number is not found in the NLS_Error_Message_Table.
1500	CRF Error
1502	CRF Error. Translation was requested, but a destinationLogicalId attribute was NOT found on either the Message or CrfActionGroup tags. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The destinationLogicalId is a required field in order to access CRF functionality. Neither the message nor the specific CrfActionGroup contained a destinationLogicalId.
1503	CRF Error. Invalid state. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The state attribute on an AlternateId is invalid. Valid states are add, delete, modify, referenced, exists, added, modified, deleted.
1504	CRF Error. NULL state attribute on AlternateId. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The state attribute on an AlternateId is required, but none was specified.

1505	CRF Error. NULL keyGroupType attribute on KeyGroup tag. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The keyGroupType attribute on a KeyGroup is required, but none was specificed.
1506	CRF Error. Invalid UUID within KeyGroup. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The UUID at the referenced message location is invalid. The UUID either does not exist on the CRF_Table or, if the first AlternateId has a state of exists, the UUID to which that AlternateId is attached does not match the UUID in the KeyGroup.
1507	CRF Error. NULL value attribute on AlternateId. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The value attrbiute on an AlternateId cannot be NULL.
1508	CRF Error. NULL sourceLogicalId on AlternateId. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The sourceLogicalId attribute on an AlternateId cannot be NULL.
1509	CRF Error. An AlternateId with a state of 'exists' must be the first and only AlternateId in a KeyGroup. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** If an AlternateId in a KeyGroup is used to index into the CRF_Table, that AlternateId must specify a state=exists AND that AlternateId must be the first AlternateId in the KeyGroup.
1510	CRF Error. AlternateId does not exist in database. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlternateId with value, sourceLogicalId and keyGroupType does not exist on the CRF_Table.
1511	CRF Error. The database record for this AlternateId shows it connected to a UUID that does not match the UUID on the KeyGroup. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].**

1512	CRF Error. The AlternateId to be deleted will be deleted by a previous AlternateId that specifies a delete. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The message contains multiple deletes for the same AlternateId.
1513	CRF Error. The AlternateId to be deleted was not found in the database. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlternateId requested for deletion does not exist on the CRF_Table.
1515	CRF Error. The AlternateId(modify) would be deleted by a previous AlternateId(delete). Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlternateId specified for a modify is also specified for deletion elsewhere within the message.
1516	CRF Error. The AlternateId(modify) would be modified by a previous AlternateId(modify). Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlternateId specified for a modify is also specified for a modify elsewhere within the message.
1517	CRF Error. The AlternateId(modify) was not found in the database. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlternateId specified for a modify does not exist on the CRF_Table.
1519	CRF Error. The AlternateId(modify) has sourceLogicalId, newValue, and keyGroupType attributes that already exists in the database. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The newValue, sourceLogicalId and keyGroupType of the AlternateId specified already exists on the CRF_Table.

1520	CRF Error. The AlternateId(modify) has the same sourceLogicalId, newValue, and keyGroupType attributes as a previous AlternateId(modify). Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlernateId specified has a duplicate elsewhere in the message.
1521	CRF Error. An AlternateId with the given sourceLogicalId, keyGroupType, and value already exists in the database. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlternateId specified already exists on the CRF_Table.
1522	CRF Error. Exceeded the per message maximum 999999 AlternateId(s) with a state of "Add". Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** Too many adds are being done in this message.
1523	CRF Error. No AlternateIds with given keyGroupType and destinationLogicalId were found in database connected to the given UUID. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The specified AlternateId does not exist on the CRF_Table.
1524	CRF Error. NULL value attribute returned from database on lookup. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** An invalid value was returned from the CRF_Table. The recor dont he CRF_Table is corrupt.
1525	CRF Error. The AlternateId to be added is being added by a previous AlternateId(add). Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlternateId specified for an add is is also specified for an add elsewhere within the message.

1526	CRF Error. The AlternateId to be added duplicates a previous AlternateId(modify). Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The AlternateId specified for an add will result in a duplicate of an AlternateId resulting fomr a modify elsewhere in the message.
1527	CRF Error. Invalid System Symbolic. Error at **CrfActionGroup[ ], KeyGroup[ ], AtlernateId[ ].** The System Symbolic is not defined to the SDR_Table.

Table 1: Error descriptions

NLS message components

An NLS error message consists of the following:

Error number
Error numbers are 4 digits long and used to identify where and why an error occurred. NLS Standards specify a length of 9 characters, but for the WMQI Enabler implementation we have set aside 5 digits for future use.

Error numbers are assigned based on the examples below:

Reserved	0000 - 0999 <-- reserved for development use.
General	1000 - 1499
CRF	1500 - 1999
Reserved	9000 - 9999 <-- reserved for customer use.

Section numbers
Section numbers are used to maintain the proper sequence of text and values within a message.

Message text
Message text is used to describe the error that has occurred or to give supporting information to a value that is placed in the message.

Space before
Space before is the number of spaces to be placed in the message string before the textual portion of the message is added.

Space after
Space after is the number of spaces to be placed in the message string after the textual or value portion of the message is added.

Text only
Text only means no value will be assigned to this part of the message.

Below is an example that employs together all NLS components:

```
<Message_Number>1005</Message_Number>
<Section>
   <Section_Number>1</Section_Number>
      <Message_Text>
      The Message cmdType was invalid.
      </Message_Text>
      <Space_Before>2</Space_Before>
      <Space_After>2</Space_After>
      <Text_Only>False</Text_Only>
</Section>
```

NLS error creation

When an error is created by MQSI, an exception is thrown to the failure path and sent to the Error Handler routine. The Error Handler routine reads the exception and converts it to the format read by the NLS_Error subflow and includes it in the MQSI_WorkArea. From there it is passed to the Log_Error subflow which contains the ProcessNLSError subflow. You can see what the Exception format created by MQSI looks like in the MQSI manual ***Using the Control Center***.

When a UserException created by the WMQI Enabler application occurs, the error information is placed within the Exception area provided by MQSI. The following fields are assigned values when a UserException is generated.

The following figure demonstrates a sample use of the UserException area:

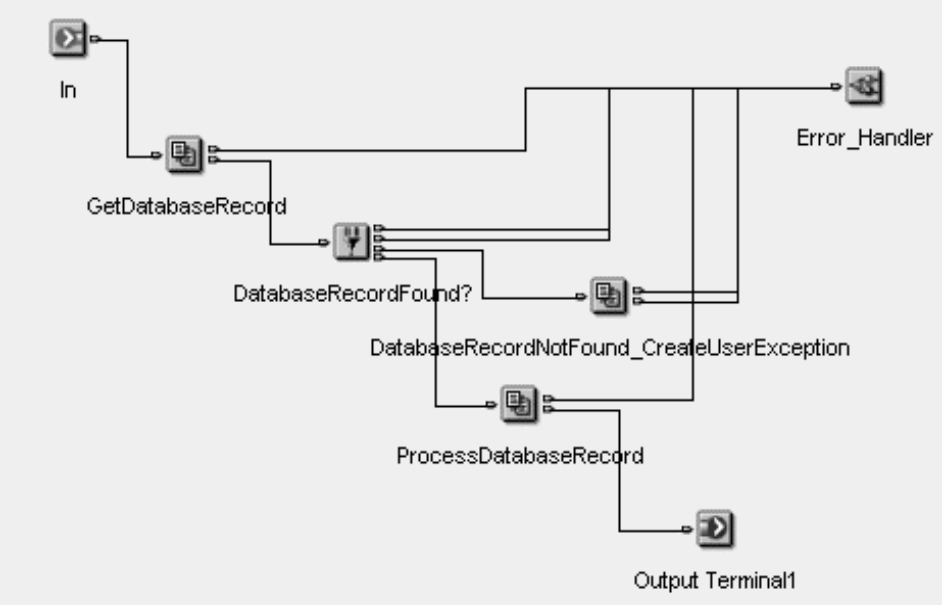


Figure 19: Sample UserException area usage.

Adding UserException values to messages

UserExceptions are most commonly placed in a Compute node immediately after a Filter node, But can be placed in a compute node anywhere in a Flow.

A label is used to identify the Flow/Subflow path and node that generated the error, as shown below:

```
SET OutputExceptionList.UserException[i].Label = 'Flow
that had the Error';
```

The Insert Type field is used to identify the type of data that is included in the Insert Text field. WMQI Enabler only supports type 2 (String) data to be place in the Insert Text field. WMQI Enabler has identified a unique Insert Type that is not used by the MQSI application to identify an exception as a UserException. This is the 999 Insert Type, as shown below:

```
SET
OutputExceptionList.UserException[i]."Insert"[1].Type
= 999;
```

The Insert Text field is used to contain text describing the error condition or a value that will be included in the message when the ProcessNLSError subflow generates the message, as shown below:

```
SET
OutputExceptionList.UserException[i]."Insert"[1].Text
= '1005';
```

Tables used in NLS processing

Language number identifies the language to be used when an error is generated. It is an additional column to the System_Status_Table.

Default installation information, platform and product version number are added to the NLS_Error_Message_Table (new to WMQI Enabler) to enhance the debugging capabilities.

Hardware platform, product version, and default language are added to the Install_Data_Table (new to WMQI Enabler).

All of these tables are found in the WMQI Enabler database.

ProcessNLSError subflow

This subflow full fills several function, which are explained below with samples.

Converting exceptions to NLS XML format

The ProcessNLSError subflow converts UserExceptions to XML.

Below is a sample converted NLS XML message:

```
<NLSErrorMessage>
  <NodeName>Flow that had the Error</NodeName>
  <Message_Number>1005</Message_Number>
  <Section>
    <Section_Number>1</Section_Number>
    <Value>475</Value>
  </Section>
  <Section>
    <Section_Number>2</Section_Number>
    <Value>55</Value>
  </Section><Section>
</NLSErrorMessage>
```

Gets NLS data from NLS_Error_Message_Table

ProcessNLSError subflow retrieves the NLS Error message from the database.

Below is a sample of the retrieved data:

```
<NLS_Results>
    <LanguageNumber>10</LanguageNumber>
    <MessageNumber>1005</MessageNumber>
    <Section><SectionNumber>1 </SectionNumber>
      <MessageText>First Part Of Message</MessageText>
       <SpaceBefore>2</SpaceBefore>
       <SpaceAfter>2</SpaceAfter>
       <TextOnly>False</TextOnly>
    </Section>
    <Section>
       <SectionNumber>2 </SectionNumber>
     <MessageText>Second Part Of Message</MessageText>
       <SpaceBefore>2</SpaceBefore>
       <SpaceAfter>2</SpaceAfter>
       <TextOnly>False</TextOnly>
     </Section>
</NLS_Results>
```

Get Language Number

Retrieves the language number from the System_Status_Table.

Below is a sample of the retrieved data:

```
<NLS_Results><LanguageNumber>99</LanguageNumber>
```

Get Default Installation Data

Gets the Default Installation data from the Install_Data_Table.

Below is a sample of the retrieved data:

```
<INS_Results>
<Hardware_Platform>N</Hardware_Platform>
<Product_Version>01</Product_Version>
<Default_Language>10</Default_Language>
</INS_Results>
```

Hardware platform
Contains information about the host system, whether it is NT, AIX, MVS, AS400.

Product version
Holds the Product version number as a single digit code.

Default language
States what the default language is set to. The language code for English is 10.

Completed error message

Once all of the information is gathered, ProcessNLSError puts it all together and creates the message to be logged to the Error_Log table in the FSE_ERRL database. You can also see the completed error message in the MQSI_WorkArea before it is removed in the Advanced_Output subflow.

Below is an example of a completed error message that will appear in the Error_Table of the WMQI Enabler database:

```
<Error>Error number N01-991005 occurred in node Flow
that had the Error:  First Part Of Message  475  Second
Part Of Message  55</Error>
```

Updating the Install_Data and System_Status tables

The NLS_Error_Message_Table, Install_Data_Table, and System_Status_Tables are updated and displayed via Hub_Only_Offline commands which are listed below:

GetNLSEntry
This command requests a particular NLS entry as stored in the Hub, which enables the user to get a response message showing what their system's NLS entry looks like in the FSE_SYSP database.

UpdateNLSEntry
This command adds or updates a particular NLS record as stored in the Hub, which enable the user to update their system's NLS entry in the WMQI Enabler database.

GetNLSEntry must be called beforehand to get the existing values in the database. The UpdateNLSEntry always does a complete replace, so data that is not included is deleted.

GetINSEntry
This command requests a particular INS entry as stored in the Hub, which enables the user to get a response message showing what their system's INS entry looks like in the WMQI Enabler database.

UpdateINSEntry
This command adds or updates a particular INS record as stored in the Hub, which enable the user to update their system's INS entry in the WMQI Enabler database.

GetINSEntry must be called beforehand to get the existing values in the database. The UpdateINSEntry always does a complete replace, so data that is not included is deleted.

GetSystemProfile
This command requests a particular system profile record as stored in the Hub, which enables the user to get a response message showing what their system's profile looks like in the WMQI Enabler databases.

UpdateSystemProfile
This command adds or updates to a particular system profile record as stored in the Hub, which enables the user to update their system's profile in the WMQI Enabler databases.

GetSystemProfile must be called before hand to get the existing values in the database. The UpdateSystemProfile always does a complete replace, so data that is not included is deleted.

Logging capabilities

WMQI Enabler provides for a number of options in regards to logging. WMQI Enabler provides a standard set of logging functions with a design that allows for additions to be made without a redesign of the logging architecture. Events are logged for every instance of an occurrence, and data analysis can be performed very easily in conjunction with these logs. In addition, customizations could be made to the logging capabilities.

Message logging

The way that message logging is implemented in WMQI Enabler requires DB2 to store the XML messages as they are being processed through WMQI Enabler. This storage is done as a means of message caching for MQSI, and to relieve MQSWF from having to carry the entire message throughout a process flow. Both XML and MQSeries Workflow messages are logged at the beginning and end of all major flows.

This logging serves dual purposes. Warehousing of the messages is done for error and event logging. The messages can be looked at or retrieved at a later time. Secondly, this allows the large XML message to be placed into DB2 storage while MQSeries Workflow takes over control of the process. When control is returned to MQSI, the message is pulled back out of DB2 for further processing.

It is recommended that the system administrator determine how long messages should be kept. Keep in mind that the information is correlated to the information that is used by MQSWF during process execution. It is recommended that a copy be made of the "cached" messages as a way of archiving the information for analysis and history purposes.

Events log

Event logging is done each time a message is processed through WMQI Enabler.

When a message is received by WMQI Enabler, whether it be a request message or a response message, the complete message is logged, including each element of the MQMD Header and the entire XML message. The Header elements are stored as their respective data types, while the command portion of the message is stored in BLOB format, to the Message_Log_Table in the WMQI Enabler database.

The following figure shows both the MQSI Message Flow, and the DB2 database table that is used:

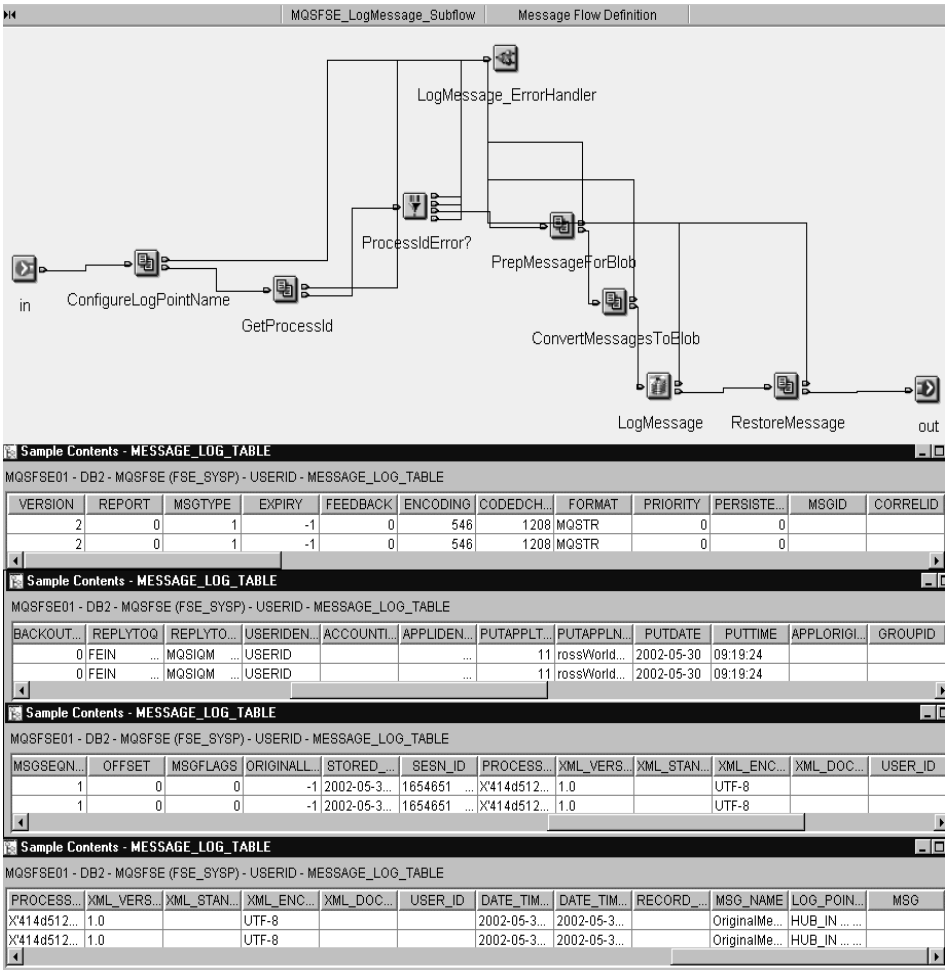


Figure 20: Event log.

Logging for history records or data analysis

Since each individual message processed through WMQI Enabler is stored in the Message log, data can be extracted at any time for analysis work. Statistical information could be gathered, either manually, or through a custom application, and the data could then be processed either manually or through the use of a custom application or vendor specific product.

Error log

When an error of any type occurs, the error message can be found in the Message_Table of the WMQI Enabler database. The entry for the message in question could be taken from the database to do any number of analysis functions on it. In addition, when an error is found internally in MQSI or MQSWF, measures are taken to document what has occurred. MQSI will place information regarding the error in the WMQI Enabler database, so that it can be used for problem determination.

MQ audit log

MQSeries provides a set of standard logs. For information on using and configuring these, refer to ***Security Server (RACF) Security Administrator's Guide***.

For information on error handling in WMQI Enabler see the ***Installation and Setup Guide***, *Appendix A*.

SDR Implemented in LDAP

Lightweight Directory Access Protocol (LDAP) states that a Directory is like a database since you can put information in, and later retrieve it. However, it is specialized. The directory structure is characterized as being designed for reading more than writing, offering a static view of the data, and providing simple updates without transactions.

The LDAP standard:

- Defines a network protocol for accessing information in the directory
- Provides an information model defining the form and character of the information
- Provides a name space defining how information is referenced and organized

Both the protocol itself and the information model are extensible.

The LDAP protocol is the vehicle for accessing the directory. It defines the operations one may perform, such as search, add, delete, modify, and change name. It also defines how operations and data are conveyed. The information model and name space are based on Entries. An entry is simply a place where one stores attributes. Each attribute has a type and one or more values:

E.g. (cn is CommonName)

cn = USERID

cn = user

mail = jdoe@us.ibm.com

IBM SecureWay Directory Server Version 3.2 is a LDAP server. It was used to create a directory structure in a format that supports the WMQI Enabler SDR design.

The LDAP custom node provided for MQSI version 2.0.1 or higher takes parameters from the node properties and the input message, performs a search on a specified LDAP server using those parameters, and constructs an output message. The output message is a copy of the input message enriched with the LDAP search results.

An export of the directory structure created in IBM SecureWay Directory and a subflow utilizing that structure are provided. The subflow is an implementation of SDR accessing the LDAP structure rather than a database. To use this subflow, remove the existing SDR subflow and put the example subflow in its place. It is entirely compatible with our existing message design and SDR/BuildDestinationList usage. The substitution of SDR implemented with a LDAP interface does not change the logical operation of WMQI Enabler.

The system interaction diagram in the following figure shows the use of the LDAP directory structure:

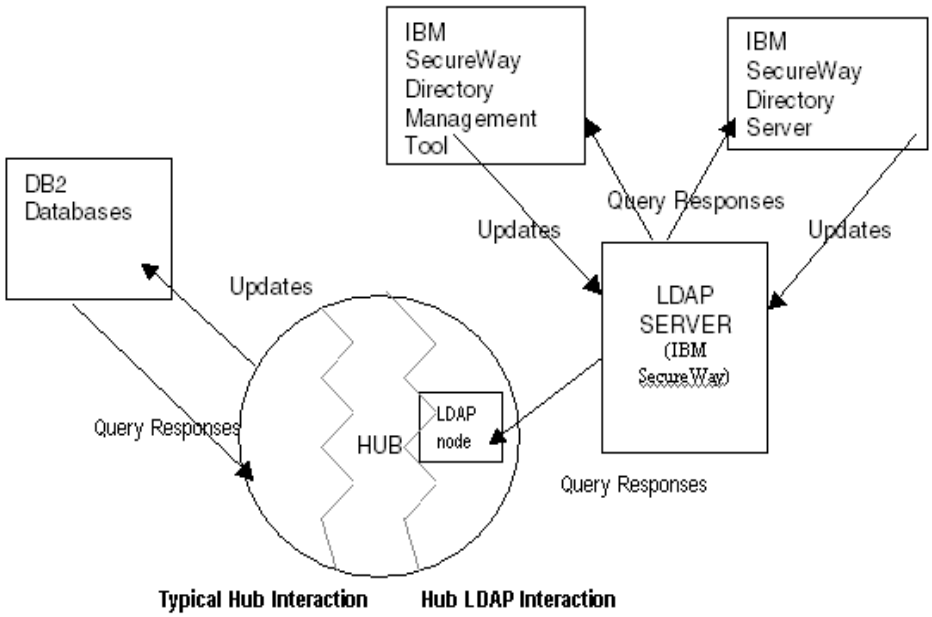


Figure 21: LDAP system interaction diagram.

LDAP directory structures are designed to exist within a static environment. These directory structures do not work well in an environment that requires frequent updates. Within the WMQI Enabler, it would make sense to use an LDAP structure

for SDR, message profiles, and system profiles. These are areas that may require only occasional updates. The directory structure would not work to implement system status, process ids, or CRF functionality. Those concepts require constant interaction with the product. WMQI Enabler has **read only** access to the LDAP structure. There is no update function available for the Hub processes. Updates must be done from LDAP applications, such as those shown above.

The following figure represents an SDR structure setup in LDAP:

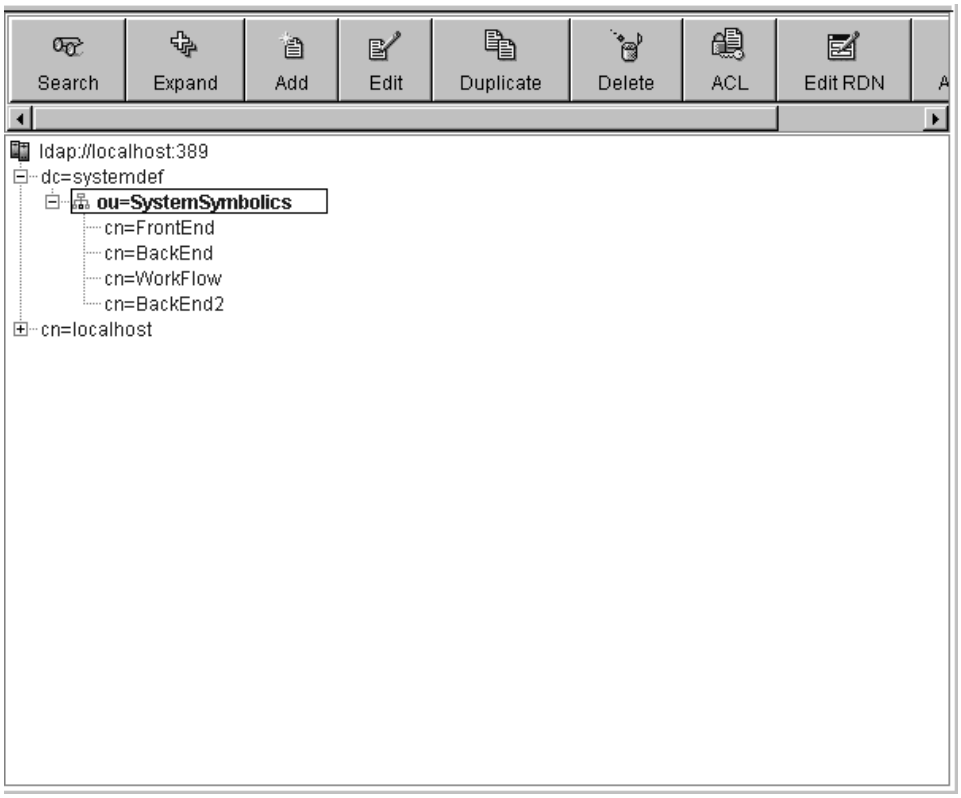


Figure 22: LDAP_SDR_System.

- The LDAP structure is built within IBM SecureWay Directory.
- The host definition name is called **systemdef**.
- The system has an object group called **SystemSymbolics**.
- The list contains **System** objects whose names represent the **System Symbolics**.

The following figure shows a system definition within the LDAP implemented SDR structure:

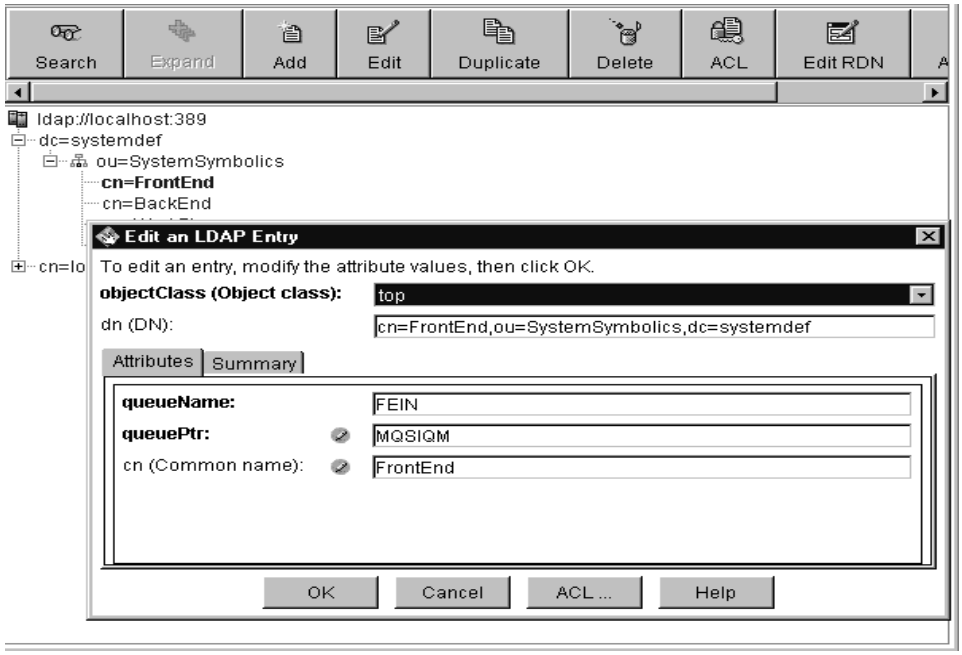


Figure 23: FrontEndLDAPSystem.

- Each **System** object contains the **queueName** and **queuePtr** of the system it represents.
- The **queueName** is the system's queue.
- The **queuePtr** is the system's queue manager.
- Within the directory structure, there are a few limitations on naming conventions. Within the same object list, there cannot be multiple System objects with the same name. However, different object lists can have System objects of the same name. Also, an object list can have different object types with the same name. For example, the object group of **FrontEnds** can have a **System** named **FrontEnd** and a **Person** named **FrontEnd**.
- To resolve the issue of having a non-System object with a system name, the query that is built specifically looks for System objects with the system symbolic names within the **SystemSymbolics** group list.

- For System objects with the same names across different object lists, the process will rely on the LDAP server administrator to monitor the directory and verify that it is properly set up. Updates and configuration for the LDAP server are out of the Hub's hands and must be properly maintained by its administrator.

The coding of the SDRFunction_LDAP subflow are system specific. The coding is directly connected to the LDAP server and structure defined on that server. Minimally, the properties of the LDAP node must be set to point at the LDAP server running the directory structure that we provide.

The properties of the LDAP node are set to the following:

Connection

Server:	Local or networked machine (mqsfse01.charlotte.ibm.com or IP address)
Port:	389 (default port)
BindDN:	cn=root (admin id)
Password:	password (admin password)

MessageTree (If these items are changed, the code must be changed also)

SearchbaseElement:	Root.XML.LDAP.SearchBase
FilterElement:	Root.XML.LDAP.Filter
SearchResultsRoot:	Root.XML.LDAP.Results

Defaults (These must be set for the node to work correctly, even if they are not used)

Searchbase:	dc=* (within all high level definitions…)
Filter:	cn=* (find all common names)

* Represents wild cards.

NOTE *LDAP support capability is provided as is and limited to Windows NT systems only.*

MQSI WorkArea

The MQSI WorkArea is constructed to enhance message access from the storage database. As messages flow into the WMQI Enabler product, the message data is stored in the WMQI Enabler MessageLog Database. The design of the product indicates that this message data is extracted from the database when required. Through the use of the MQSI WorkArea, message data that is consistent over the life of the message is retained in the WorkArea. This WorkArea provides a global, variable space that is separate from the message structure.

The WorkArea is basically an information wrapper that is placed around the message. It is a "work area" to pass parameters between MQSI nodes. It is also capable of holding multiple messages for the purpose of forming another message from the original message while retaining the data from the original message. It functions similarly to a "shopping bag", where data items are placed that will be used within the message flow.

For instance, the AdvancedInput_Subflow contains an AddWorkArea_Subflow that has a compute node that is utilized to modify the message with the addition of the WorkArea. In the same fashion, the AdvancedOutput_Subflow contains a RemoveWorkArea_Subflow that has a compute node that restructures the message to remove the WorkArea plus select the proper message to be delivered to the appropriate destination.

WorkArea tags

A series of tag names are employed to accomplish the features of the WorkArea. These tags are created and used at different points in the numerous MQSI subflows. Sample tag names are as follows.

<MQSI_WorkArea>
: Functions as a high level tag name that defines the global, variable space

<Message_Group>
: Functions to provide every message that the process may be interacting with.

<OriginalMessageOffset>
: Functions to provide the first occurrence of a MessageItem displaying the original message that started the particular flow.

<MessageItem>
: Functions to provide the message type, XML or MQSeries Workflow, the process identification, and the message name. This MessageItem thus serves to provide the beginning of the message.

<WfMessage>
: Functions to provide the high level tag name for a MQSeries Workflow message.

<Parameters>
: Functions to provide an execution list that resolves multiple or single system symbolics.

<Results>
: Functions to provide the results of the executions found in the *<Parameters>* area supplying system symbolics, system status, queue managers, queues, and status of queues.

<HistoryList>
Functions to provide a history of the result list as new processes are run. The *<HistoryList>* has a *<Status>* to indicate whether the process succeeded or had an error.

<BuildDestinationList>
Functions to provide the *<Parameters>*, *<Results>*, and *<HistoryList>* for the definition of the destination queue and queue manager and the reply to queue and queue manager.

<RemoveWorkArea>
Functions to remove the message tag, found in *<MessageItem>*, and route it to a queue.

<LogMessage>
Functions to provide the *<Parameters>*, *<Results>*, and *<HistoryList>* for use in the LogMessage_Subflow where various attributes are promoted to the level of the flow.

<LogPoint>
Functions to provide a correlation to the names the MQSeries Workflow uses to refer to the messages.

<LogMessageOffset>
Functions to point to the offset of the message that is to be logged so independent data may be pulled, such as MessageId or SessionId.

<LogPath>
Functions to provide the actual content of the "blob" section of the message log that may contain the entire WorkArea, a single message, or some wild cards with select statements.

<ErrorList>
Functions to provide a log of errors that will be logged into the Error_Log_Table.

<TraceArea>
Functions to provide the start and end of trace activity. Can contain multiple items and is logged to a trace database.

<MessageProfile>
Functions to provide a MQSeries Workflow parameter list that represents the information in the MessageProfile database.

<SystemInteractionList>
Functions to provide a list of the systems that the message needs to access.

<InteractionProblemFlag>
Functions to locate backup systems or to report if there is a problem with the system interaction.

<Session>
Functions to provide session data for session authentication.

<SystemProfileArea>
Function to provide the system profile information that is contained in the System Profile database.

<CRF_WorkArea>
Functions to provide the information that is required at various points for the cross-reference functionality.

Complex Business Processes Support (Update for Complex Use Cases)

Communication between WMQI Enabler and Workflow was implemented so that WMQI Enabler must be the initiator of the process. Complex Processes Support enables Workflow to initiate the communication with WMQI Enabler. A Workflow process sends a request message to WMQI Enabler. Workflow is unable to send a message in an WMQI Enabler format (such as AddParty), but it is able to specify the name of a message type. Templates that correspond to message types can be stored in an WMQI Enabler database. These templates can be skeletons of the basic message format or contain as much common data as the user requires. Along with the template name, Workflow sends the message fields and data that are to be added or manipulated within the template. Upon receiving the Workflow request to process a template, WMQI Enabler retrieves the template, performs the requested field changes and additions, creating a fully formatted WMQI Enabler message, and sends this message to HUB_IN for processing. The message behaves as if it came from any FrontEnd requesting system. It requires a message profile, SDR entries, a valid session ID if Session Validation is required, system profiles if Interaction Check is required, and valid CRF information if the message specifies CRF processing, to name a few examples. After the template message has completed processing, the final response message is sent to the Workflow process that made the request.

Using this functionality

To use this functionality, the message template table must be populated. Three example messages are provided: AddPartyInsert, AddPartyUpdate, and AddPartyDelete. MQSIGet can be used to place these messages on the TEMPLATE_IN queue. Currently, these messages specify FEIN as the response queue.

A process called Supervisor is provided to kick off a message to WMQI Enabler. To use this process, import it (Supervisor.fdl) into Workflow runtime, open the Workflow client, and select the supervisor process using the right mouse button. Choose “Create and Start Instance”. A window will open requesting input data. Because input data is not required, click OK to start the process. Workflow will then send a message to MQWF_OUT requesting to use the AddParty template. A valid message profile must exist in the database for AddParty.

In addition, a test suite is provided for handling the AddParty processing. This suite is contained in a .zip file called IAACBP.zip, and is called IAACBP_Testsuite.xml. Its primary function is to provide a listener for the backend system and to send the sample response. The test suite also sets the Stored Message Template and then sends a message to Workflow to initiate the Supervisor process within MQSeries Workflow (as an alternative to manual processing as mentioned in the previous paragraph). The Supervisor process then sends a message to WMQI Enabler that calls out the users Stored Message Template message for processing.

Once processing is done, check the Message_Log_Table to verify that the proper path has been taken. There should be a MQWF_OUT for the original message, HUB_IN for the entry of the AddParty, MQWF_OUT, HUB_RWF_IN, or HUB_R_IN logs for the add party, depending on whether or not it processed through Workflow, a HUB_RWF_IN entry for the response to the Supervisor process, and a single MQWF_END if the AddParty went through Workflow.

Synchronous versus Asynchronous Processing

WMQI Enabler processes request messages that require a response (synchronous) and those that do not require a response (asynchronous). The specification is made by the requesting system which must set the MsgType field of the MQMD header. A value of MQMT_REQUEST identifies the message as being synchronous. Values of MQMT_DATAGRAM and MQMT_REPLY indicate that message is asynchronous.

Synchronous Processing Requirements

A request message that is synchronous must provide information indicating where responses need to go. This can be done in two ways: populating the MQMD header ReplyToQ and ReplyToQMgr fields, and/or providing a sourceLogicalId attribute in the message. If both exist, the reply to information from the MQMD is used to send back the response and any error messages that occur.

Upon sending the response message, (HUB_R_IN for non-Workflow processing, MQWF_OUT for Workflow processing) KillProcess is used to deactivate the process id generated for this transaction.

Asynchronous Processing

In the event that a front end requesting system does not require a response, reply to information and the sourceLogicalId are not required.

NOTE: The sourceLogicalId is required if this message intends to use system interaction check. Upon failure of system interaction, the sourceLogicalId is required to pull out the store forward flag in the system profile.

At the end of HUB_IN, if the message is asynchronous and not going to Workflow, KillProcess is used to deactivate the process id generated for this transaction.

For asynchronous transactions that use Workflow, a Workflow process must be configured to not send the final response to the requesting system. In the final activity of the process, the NoDestination flag must be set to “True”. This will tell MQWF_OUT to not send out the response. AddPartyAsync.fdl is provided as an example.

Error Message Destination

In an error situation, the generated error message would be sent to the reply information provided by the message. If that reply information does not exist, the message is routed to the system specified in the sourceLogicalId of the message. If the sourceLogicalId doesn’t exist, the message is routed to a default system symbolic of “ErrorSystem”. The queue and queue manager associated with this symbolic are configurable in the SDR table. ErrorSystem is giving a default queue and queue manager of “ERROR_BUCKET” and "MQSIQM".

A system may specify how it would like to use the default error system symbolic. Within the system profile is the field ERROR_MSG_DEST. This field has five possible values that WMQI Enabler supports: ‘ ’, ‘SOURCE", “DEFAULT", “BOTH", and “NONE". ‘ ‘ indicates that the field is empty and the error message will attempt delivery to the source system. SOURCE indicates that the error message is to go to the source system. DEFAULT indicates that error messages for this system

should go to the default error system of ErrorSystem. BOTH indicates that the error messages should go to the source system and the default error system. NONE indicates that error messages should not be sent anywhere.

Within the PUB/SUB functionality, specific messages can be set to PublishOnError, based in the message profile. In this case, the request message will be published with error information if an error occurs.

Communications Between Remote Systems

The reply to information specified in request messages that enter WMQI Enabler for processing can specify remote queue managers that WMQI Enabler does not have visibility to WMQI Enabler and this remote system would be communicating through a series of channels and remote queue definitions.

For example, if we are using a system FE on queue manager FEQM, and this system is communicating with the WMQI Enabler queue manager MQSIQM, both queue managers would have a pair of sender and reciever channels possibly called FEQM.TO.MQSIQM and .MQSIQM.TO.FEQM.

System FE would have a transmission queue possibly named MQSIQM.XMIT, and a remote queue definition for HUB_IN that would identify the remote queue name as HUB_IN, the remote queue manager as MQSIQM, and the transmission queue of MQSIQM.XMIT. When sending a message to WMQI Enabler, FE would place it's request on the remote queue definition of HUB_IN located on queue manager FEQM.

Ideally, MQSIQM would have a smiliar set up, having a transmission queue possibly named FEQM.XMIT, and a remote queue definition of FE_RESPONSE, that would identifiy remote queue name of FE_RESPONSE, the remote queue manager of FEQM, and the transmission queue of FEQM.XMIT. WMQI Enabler would look at the reply to information provided by the request message. If that reply to information is of queue FE_RESPONSE and queue manager of MQSIQM, WMQI Enabler would see the local remote queue definition FE_RESPONSE on MQSIQM, send the message to there, and because it is a remote queue definition, the FE system would receive its response on the FE_RESPONSE queue on queue manager FEQM.

However, if the reply to information provided by system FE specifies a queue name of FE_RESPONSE and the queue manager FEQM, MQSeries used by WMQI Enabler will not be able to resolve that queue manger name, because it does not have visibility to it. The message must be sent to the remote queue definition on the local WMQI Enabler queue manager of MQSIQM. For this functionality, WMQI Enabler provides two fields in the message profile: UseHubQMgrAsReplyFlag, and HubQueueManager. When the flag is 'True', and the hub queue manager is populated, every response situation, (error message issued by WMQI Enabler, hub only command response generated by WMQI Enabler, and the actual response

from the back system being sent through WMQI Enabler), uses the hub queue manager as the destination queue manager, and ignores the queue manager specified in the reply to information of the request.

In the example, FE would send in a request message of AddPerson with reply to information of FE_RESPONSE on FEQM. The message profile for AddPerson would indicate that the HubQueueManager is 'MQSIQM', and the flag UseHubQMgrAsReplyFlag is 'True'. When the response from the back-end system is being sent to FE, WMQI Enabler would send it to the "reply to" queue FE_RESPONSE located on the hub queue manager MQSIQM. This would cause MQSeries to send the message to the remote queue definition of FE_RESPONSE on MQSIQM, causing the message to properly go to FE_RESPONSE on FEQM.

Chapter 7
WMQI Enabler and MQSeries Workflow

A key feature of WMQI Enabler is the ability to move business processing into the WMQI Enabler product. This feature allows for customization of the business process to become a real part of the enterprise without the need to modify existing applications. Coupled with MQSI, MQSeries Workflow provides supervision of these business processes and features:

1. Moving the business transaction requirement from desk top to completion.
2. Supports transaction logging and recording of history.
3. Rapid updates of business transaction processes.

Manipulating workflows

MQSWF provides a Buildtime environment for visually designing and creating the graphical diagram of the business transaction process flows. The use case and business transaction sequence diagram plus the system interaction diagram are helpful and act as a guide in this design to more precisely lay out the graphical representation of the business process. The MQSWF modelers should be familiar with the XML message being processed since data flows are part and parcel of MQSWF. In addition, construction of the MQSWF data containers is key to a successful transaction.

Any number of workflow templates can be created to satisfy the needs of the existing applications, and these workflow process templates are a direct result of the business transactions that are being modeled. Any previously designed workflow process templates may be used as a starting point, and often, with only minor modifications, can be customized to achieve a new workflow. Of course, any workflow process templates that are determined to be not applicable can simply be removed or deleted from the Buildtime component of MQSWF. Once satisfied with these business transaction flows, they can be imported into an MQSWF Runtime environment for testing and implementation into a production environment.

Workflow considerations

WMQI Enabler uses the MQ Workflow process flow to control the processing of messages. In order to activate the required MQSWF process flow, the XML message must be "wrappered" in a MQSWF XML header that provides MQSWF with the information required to initiate MQSWF processing. MQSI is used to accomplish this action.

The MQSI XML to MQSWF XML "wrappering" process copies control information into the MQSWF header. Additionally, the message profile is used to pre-define the content of the XML message that should be included in the workflow container. This "Message Profile" gives the ability to have custom content in the workflow container on a message by message basis. The contents of the XML message are stored in a database. This action allows access to the XML structure when required but maintains a lightweight message container for MQSWF. Once this "wrappering" process is completed, the newly formed MQSWF XML message is put to the MQSWF XML queue so that the appropriate MQSWF process template can be initiated.

Once execution begins within the MQSWF process template flow, the response to the original XML message request can be obtained by executing the steps within the predefined process flow. There are four types of activities that can take place:

1. Execution of a hub function.
2. Execution of an external application via an adapter.
3. Execution of an internal workflow function.
4. Execution of a workflow client related activity.

The workflow functions and client activities do not require special support within WMQI Enabler, but they can be used to provide powerful features and function as a part of the solution to an WMQI Enabler implemented use case.

In the case where external workflow process or workflow clients are used, the process can either access the XML content from the message cache directly, although they will have to parse the content, or they can rely on the content available in the workflow container. In either case, the process is interacting directly with workflow and as such must follow the MQSWF API conventions.

In the case where an activity is intended to interface with an external application via an adapter, the MQSWF flow releases the message to a pre-defined queue. In the Model Office implementation that queue name is "MQWF_OUT". Once the message is on the queue, MQSI takes the message off of the queue and processes it using the message flow "MQWF_OUT_Flow".

The "Message Profile", attached in *Appendix C* of the ***Installation and Setup Guide*** and described in *Chapter 4* of this manual, is a DB2 table that must be built for each message. The options provided in the Message Profile database table will guide how the message works and flows within WMQI Enabler. Special attention

should be given, however, to the "Parameter_Name" and "Parameter_Path" fields. In these fields, any message data that is required in the workflow template initial activity can be specified.

In the workflow templates found in the WMQI Enabler product, the name, "WFTransition", has been specified in the "Parameter_Name" field, and the "Parameter_Path" is specified as XML.(XML.tag)”Message”.(XML.tag)”COMMAND”.(XML.tag)”AddPartyResponse” .”(XML.attr)”cmdStatus”.

The "Parameter_Path" field will allow the product to go that spot in the XML message, returning values of "ok" or "notok", and assign them to the "Parameter_Name" of "WFTransition".

This setup, then, allows the use of "WFTransition" to define the transition conditions within MQSWF. Multiple fields may be established in the Message Profile database for use in the initial activity. The path in the message would then need to locate the value for use, e.g. XML.(XML.tag)”Message”.(XML.tag)”COMMAND”.(XML.tag)”AddPartyResponse” .(XML.attr)”cmdStatus”.

The settings or values found in the Message Profile only apply to the initial activity. The use for such values as "RequestedParameters" or "FieldChanges" (see definition in the ensuing comments) will have to be defaulted in the containers, either input, output, or both, if they are used in the messages for later activities. The standard techniques for defaulting values, as outlined in the MQSWF documentation, should be employed for that purpose.

The activity must use the WMQI Enabler container structures defined for request and response processing in a workflow activity. These containers allow the activity to update content and define the returning container definition on the request side. The response container must match the definition specified in the initial request. Further, the request defines the name of the response container so that MQSI knows how to name the response container when processing in the adapter is completed and the response is returned to MQSWF.

While the input and output data containers can have varying data within them, the MQSI messages require certain fields in order to process the data through the MQSI flows. The described data may be named according to user's discretion, but the field names, if utilized, may be found in the following specification. In addition, MQSI will look for the exact name for the template data structures, which represents what is pre-defined in the MQSI flows and may be found in the FSE_WFCO table. The template data structures also anticipate multiple level nesting of the data, and that nesting scheme is pictured with the indenting in the ensuing comments. The pictured levels found in the nesting of data is also required for MQSI to recognize the data fields and to employ them for message processing.

The template data structures are described in the following comments:

ProcessTemplateExecute

The data described here is necessary to process the initiation of the workflow process template. The message is sent from MQSI to MQSWF to start the process. The data for this message flow is:

1. **CommonArea** made up of the following data elements:

   SessionId: This element is a variable length string that holds the id of the session that MQSI has assigned to a hub connection.

   ProcessId: This element is a variable length string that holds the id of the process that MQSI has assigned to the message type.

   OriginalMessageId: This element is a variable length string that holds the id of the message that started the workflow process.

   MessageId: This element is a variable length string that holds the id of the message that MQSI has received. This field is only used by MQSI when the MessageName data element of the OutGoingMessageArea is not provided (See the following comments for definition of the MessageName and OutGoingMessageArea under ActivityImplInvoke).

   Publish: In the event that a PUBLISH is required, this element is a variable length string that holds topic.

2. **SystemInfoArea** made up of the following data elements:

   System X: This data element is an array of elements where "X" is the number of system(s) for which MQSWF is expecting to receive information.

      Symbolic: This data element is a variable length string that holds the system symbolic which is used to retrieve routing information from the Symbolic Destination Routing (SDR) table.

      ActiveFlag: This data element is a variable length string that holds the system active flag that is a Boolean expression that indicates whether a system is ready to receive messages.

3. **DynamicParametersArea** made up of the following data elements:

   Parameter:

      Name: This data element is a variable length string that holds the name of one parameter (a data element).

      Path: This data element is a variable length string that holds the path that is used to retrieve the value for the parameter from the message, e.g. XML.(XML.tag)”Message”.(XML.tag)”COMMAND”.(XML.tag)”AddParty Response”.”(XML.attr)”cmdStatus”. The parameter can also be pulled from the MQMD header, e.g. MQMD.MsgId

ActivityImplInvoke

This set of data elements is required to request a set of services from MQSWF. Simply stated, it starts an activity.

1. **CommonArea** Same data elements as seen in the ProcessTemplateExecute set.
2. **OutgoingMessageArea**

   MessageName: This data element is a variable length string that holds the name to be given to the out going message (request message) so that it can be referenced later.

   ProcessReplyFlag: This data element indicates that a reply to the System that started the Wrokflow is required. Generally, this element is set to 'True' when the workflow is ready to respond to the originator.

   FieldChanges:

   Change X: This data element is an array of elements where "X" is the number of the field to which changes will be applied.

   Path: This data element is a variable length string that holds the path to a changing field, i.e. Message.destinationLogicalId.

   NewValue: This data element is a variable length string that holds the new value to place in the changing field.

   NoDestinationFlag: this data element is used in conjunction with the Publish flags to control the routing of outbound messages. When set to 'False' the outbound message will not be routed to destinationLogicalId(s). Using a combination of Publish and NoDestinationFlag it is possible to publish a message without sending that message to any specific destination(s).
3. **IncomingMessageArea**

   MessageName: This data element is a variable length string that holds the name to be given to the incoming message (response message) so that it can be referenced later.

   WorkFlowDataStructureName: This data element is a variable length string that holds the name of the data structure (established in MQSWF Implementations of Data) that MQSWF is expecting to see in the container section of the ActivityImplInvokeResponse message, i.e. ActivityImplInvoke001.

Requested Parameters:

Parameter X: This data element is an array of elements where "X" is the number of the parameter MQSWF is requesting to be sent.

Name: This data element is a variable length string that holds the name of one parameter that will appear in the DynamicParametersArea of the ActivityImplInvokeResponse message.

Path: This data element is a variable length that holds the path to be used for retrieval of the value for the parameter from the message.

ActivityImplInvokeResponse

This set of data elements is sent from MQSI to MQSWF as a response to the ActivityImplInvoke set of data elements. It represents the response from the target applications.

1. **CommonArea** Same data elements as seen in the ProcessTemplateExecute set.
2. **SystemInfoArea** Same data elements as seen in the ProcessTemplateExecute set.
3. **DynamicParametersArea** Same data elements as seen in the ProcesTemplateExecute set.
4. **ErrorInfoArea**

   Error: This data element is a variable length string that holds information describing the error that MQSI needs to send to MQSWF in the ActivityImplInvokeResponse or ProcessTemplateExecuteResponse message.

ProcessTemplateExecuteResponse

This set of data elements is utilized to signal the end of the process and return such a message to the front-end application. This set of data is used by an activity that precedes the sink node found in MQSWF so that the message may be returned to the front-end application.

1. **CommonArea** Same data elements as seen in the ProcessTemplateExecute set.
2. **ErrorInfoArea** Same data elements as seen in the ActivityImplInvokeResponse message.

While the name of the data structure in MQSWF may be specified by the individual WMQI Enabler product user, the "instance" of the data structure must coincide with the templates in the preceding comments for them to be used.

The following figures show a data structure that is nested within another data structure:

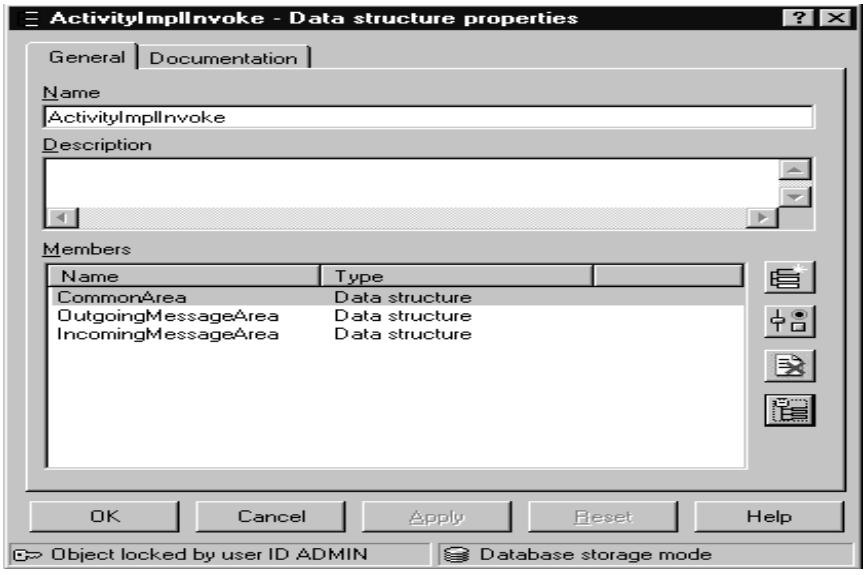


Figure 24: ActivityImplInvoke is the "instance" of this data structure.

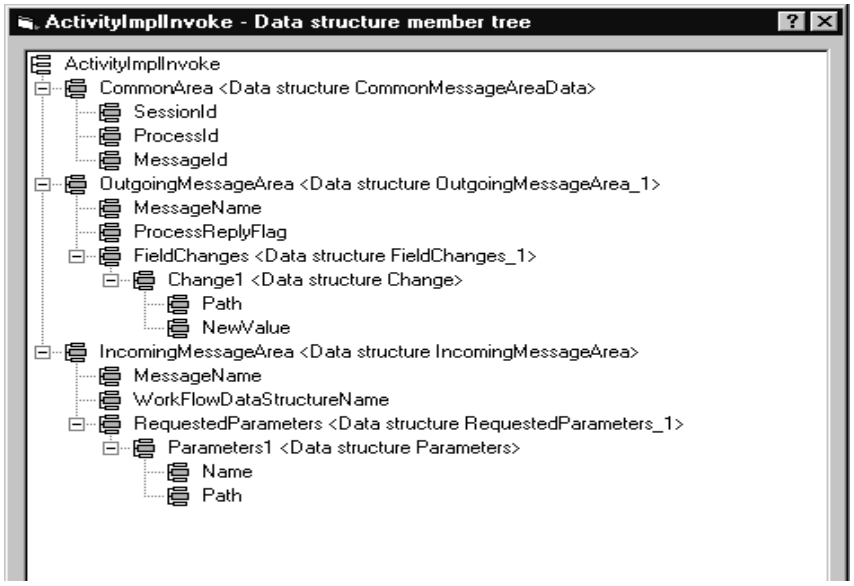


Figure 25: Data Structure for ActivityImplInvoke area.

It has a user specified data structure of "CommonMessageAreaData" that contains the data elements shown. "CommonArea" is the requirement of the MQSI message flow.

It is necessary that the naming conventions between the Body Category in the XML message, as a default value, or the name of the workflow process found in the Message Profile and MQSeries Workflow be consistent. It is required to know when the workflow process template has completed or ended. This can be accomplished by having a signal come from the sink node in the process template, since that sink node marks the end of the process. It is also required to send a message back to the source system with a response so some other option had to be created. It is now necessary to have a separate activity in the workflow process template for the purpose of responding to the source system of the message and to mark the end of the process. This last activity should immediately precede the sink node.

Modifying a data structure in a workflow

The MQSeries Workflow data structures that are to be used in the MQSWF data containers must maintain consistency with the underlying MQSI flows that support WMQI Enabler.

The MQSI flows support:

1. A message sent from MQSI to MQSWF to start a new occurrence of a Workflow process.
2. A message sent from MQSWF to MQSI so that MQSWF can request a service to be completed by MQSI. Generally, this is an activity implementation.
3. A message from MQSI to MQSWF to supply a response to the request for service.
4. A message sent from MQSWF to MQSI to indicate the process is complete and may be terminated.

MQSWF now has the freedom to indicate the data that is required for the message to process. The message also works in harmony with the Message Profile Database that is established for each message and the System Profile database that is built for each system attached to WMQI Enabler.

The WMQI Enabler architecture allows the business analyst the freedom to define the container required to support the process flow without requiring programming changes to the underlying WMQI Enabler architecture, within the constraints of the MQSWF product. The user specifies the required content in the message profile database, and this content and the predefined WMQI Enabler content are combined to build the container that becomes available in the source node of the process graph. The user then uses the predefined container structures to define the updated data to be sent as a part of the activity request message and the data

to be included in the response message. In both cases the analyst is free to define the specific content required to support the activity without requiring changes to WMQI Enabler, within the limits of the WMQI Enabler container architecture.

The user may also refer to the MQSeries Workflow manuals, but a user could simply double-click on the predefined data structure and then modify the data structure. Care should be taken to ensure that any changes remain within the boundaries described in this manual. The user will need to Apply, Save, Verify, and Re-export for the changes to be accepted.

For more information see the ***IBM MQSeries Workflow: Getting Started with Buildtime***, *Chapter 3 Creating a process model, Defining data structure section.*

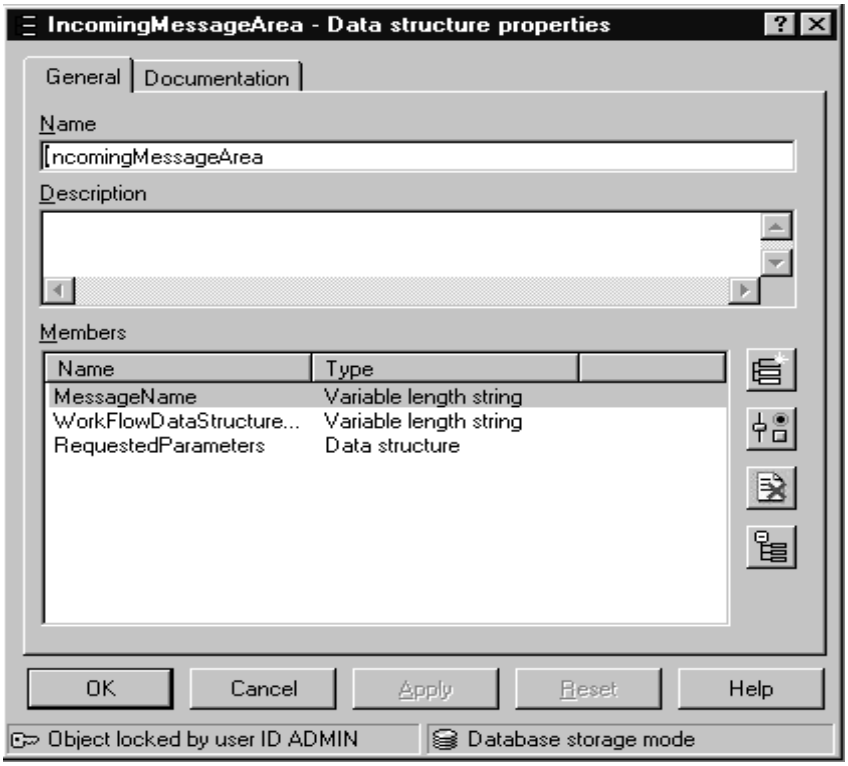


Figure 26: Data structure properties.

Adding a workflow

Samples of the use cases, sequence diagrams, system interaction diagrams, and MQSWF templates are provided in the ***Model Office Reference Manual***. However, the WMQI Enabler user may have a business need to add a new workflow to correspond to a new use case. This action will depend upon the needs and requirements of the individual business. The details of creating a new workflow are not covered in this manual; they are, however, discussed in the WMQI Enabler Planning Guide, as each new workflow is considered a new implementation for WMQI Enabler.

For more information, please see the ***IBM MQSeries Workflow: Getting Started with Buildtime***, *Chapter 3 Creating a process model.*

In accord with the WMQI Enabler architecture, the data structures, programs, and queue names will need to be verified with the MQSI and XML teams as this new workflow template is built.

Modifying a workflow in MQSWF

If the WMQI Enabler user wishes to modify an existing workflow, such as adding a new program/business rule to a claim process, then they may use the current workflow (found in the Model Office Reference Manual), and a new program (Implementation area of MQSWF Build-time) and a program or process activity node (Processes area of Buildtime) may be added to affect such a change. This new program or process activity, of course, would have the proper transition codes reflecting the new business rule to control the flow of work in the changed MQSWF workflow process. WMQI Enabler provides access to any content contained in the XML message being processed. This content forms the basis for the transition conditions that are coded within the workflow process. This is different than previous releases where a static Transition code was included in the data structure. In the previous release, this required verification from the XML and MQSI teams. (Transition codes may also come through the adapters to the external systems as return codes from those systems). Now, since any content is readily available, the analyst is free to setup the transition codes as they see fit. Please note that in some cases this implies that there will be additional setup in the message profile database table, so that the required content is available upon entry to workflow.

The WMQI Enabler user would then need to Apply, Save, Verify and Re-export for these changes to be utilized. MQSeries Workflow also provides Change Control for modified processes. This feature will allow the user to select a date in which the modified process will be activated. The date control feature may be found on the General Tab of the Process. The date control feature also helps to ensure that MQSWF Buildtime and MQSWF Runtime remain in proper synchronization.

For more information see the ***IBM MQSeries Workflow: Getting Started with Buildtime***, *Chapter 3 Creating a process model.*

The Process Tab, which is shown on the upper right of Figure 18, is where "processes" are defined.

The "AddParty" process is shown in detail; it may be added to other processes as a process node.

In the following figure, the process activity or node is indicated in the middle column and is the third icon down in the column:

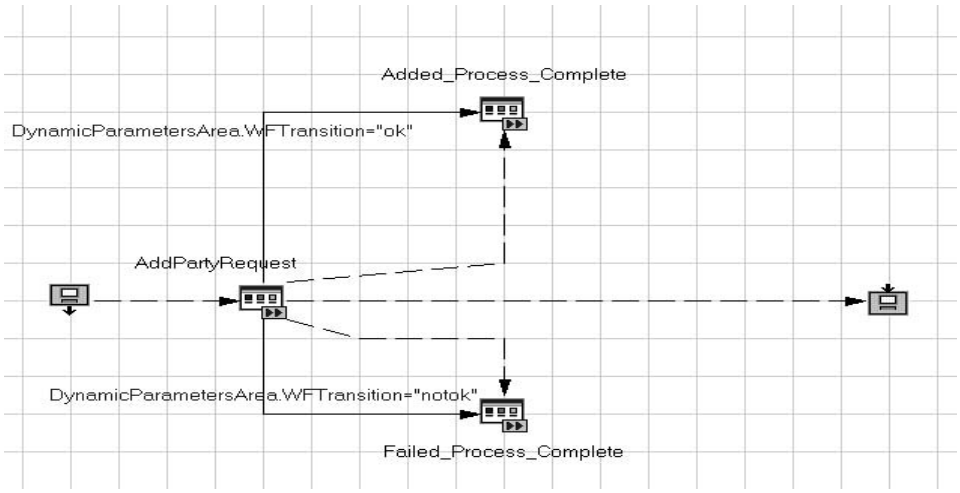


Figure 27: MQSWF AddParty.

Removing a workflow

Removing a workflow requires coordination of all parties affected by the removal of a business process. Notification should be sent to both business parties and technical parties to discuss the impact, as this business process would not be used again. Following consensus on the removal of the workflow process template, an effective date for the removal should be determined, and the workflow terminated on that date. To remove the workflow process, reference the MQSeries Workflow manuals.

For more information see the ***IBM MQSeries Workflow: Getting Started with Buildtime***, *Chapter 1 Introducing Buildtime*.

Using program clients

The MQSeries Workflow Client starts processes and monitors their execution. The Administration Utility administers the system and the Program Execution Agent invokes application programs that are used in the workflow. With an MQSeries Workflow Client, a process may be started, and a work list used to manage work items. MQSeries Workflow offers a standard MQSeries Workflow Client that is based on API's. This Client is named Runtime.

MQSeries Workflow offers API's to support the interaction between the MQSeries Workflow server and client components. In addition, the API's may be used to invoke applications that are needed for workflow tasks. Using the client API's, custom clients may be built. For example, an MQSeries Workflow Client may be built so that users may manage their work items.

For more details on the MQSeries Workflow Clients, please refer to the ***IBM MQSeries Workflow Programming Guide***.

Modifying program names & queue names

In accordance with the WMQI Enabler architecture, any modifications to a program name or queue name should be discussed with the XML and MQSI team to verify any impacts or changes that may need to be acknowledged by them as well. See the figure, below.

For more information see the ***IBM MQSeries Workflow: Image and Workflow Library***: *MQSeries Workflow Concepts, Installation, and Administration. Chapter 3 MQSeries Workflow Topology and Design Issues, Queue Manager section, Snapshot - of queue section.*

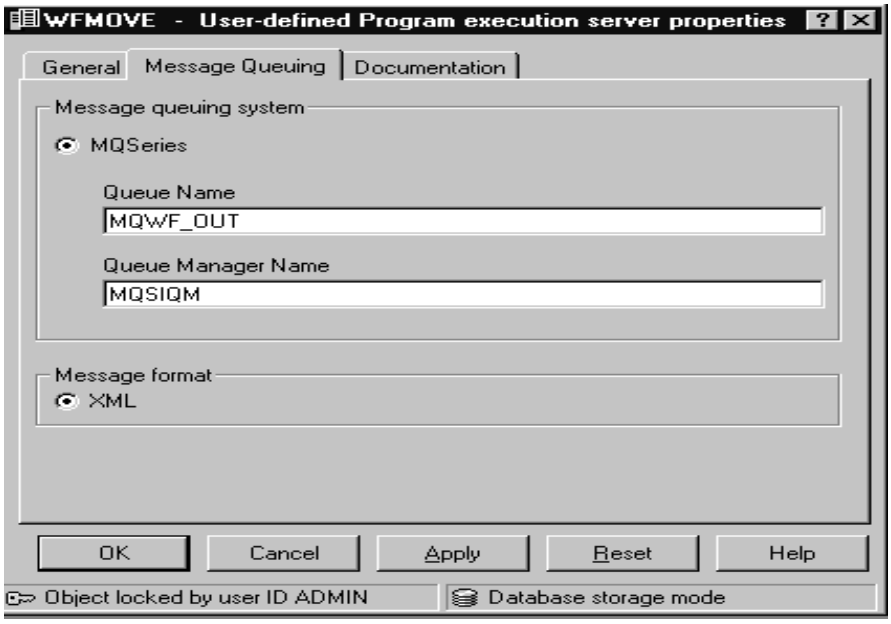


Figure 28: Modification to a QueueName.

Generic workflow samples

WMQI Enabler ships with a working set of MQSeries Workflow Processes included in the **MQSFSEWorkFlowProcesses.fdl**. Many of the processes are used by specific use case/test case scenarios. All of the processes in the FDL can be used by any messageType.

Message Profiles allow the workflow process used by a messageType to be different that the messageType name. This new feature allows any workflow process to be used by any messageType.

Several processes can be employed for general purpose application. These general workflow processes can be used as-is and are also supplied as sample, generic template from which more advanced and customized processes can be constructed.

These generic sample workflows are described in the following sections.

SetDestinationIDM workflow process template

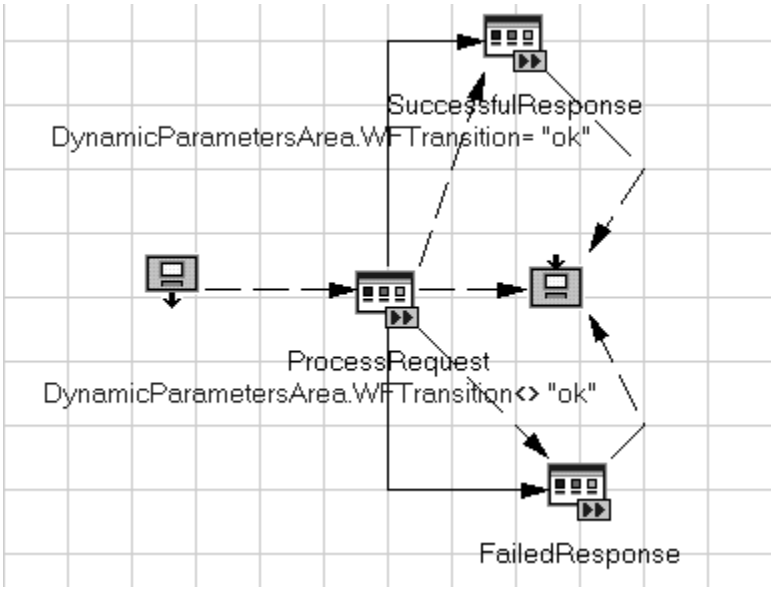


Figure 29: SetDestinationIDM workflow process template.

Verbal explanation

The SetDestinationIDM process uses information provided to the process to route an incoming message to the supplied destinationLogicalId. The response returned from that destination is checked and routed to the next activity based the value of the response message's cmdStatus (a part of a message). The response message is then routed back to the originating System. The process ends.

Recommended use

This workflow template can be used by any messageType that conforms to the IBM IDM (that is, uses cmdStatus to indicate message response status).

SyncAndPublishIDM workflow process template

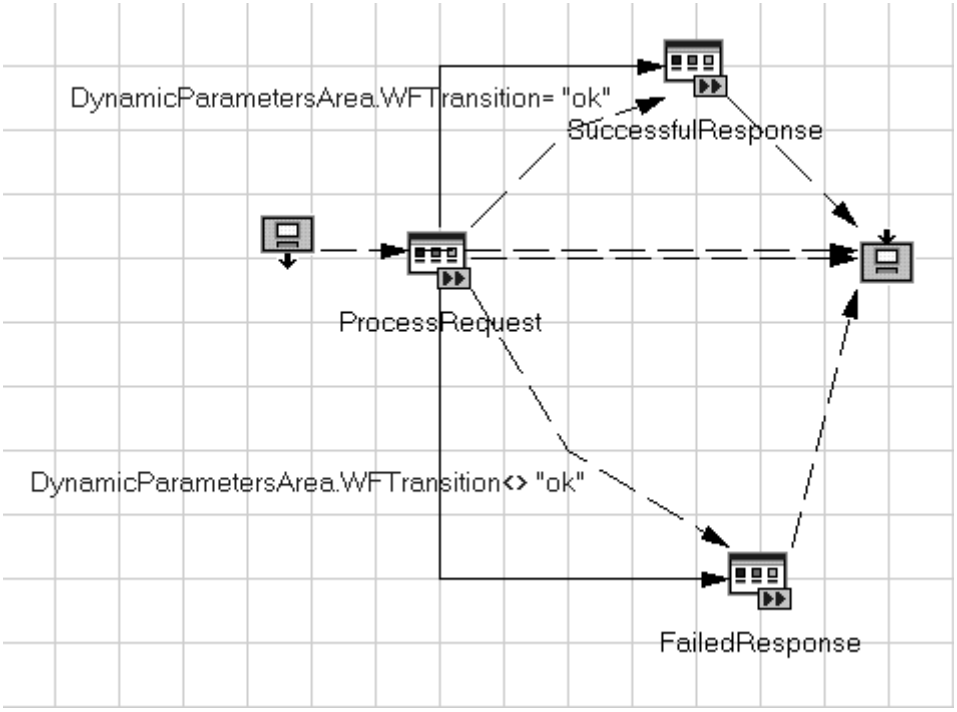


Figure 30: SyncAndPublishIDM workflow process template.

Verbal explanation

The SyncAndPublishIDM process uses information provided to the process to route an incoming message to the supplied destinationLogicalId. The response returned from that destination is checked and routed to the next activity based on the value of the response message's cmdStatus. The response message is then routed back to the originating System. The original message is then sent to WMQI Enabler for possible publishing to the supplied PublishTopic. The process ends.

Recommended use

This workflow can be used by any messageType that conforms to the IBM IDM (that is, uses cmdStatus to indicate message response status) that wants a Publish performed on message processing completion.

SyncTwoBackEndsIDM workflow process template

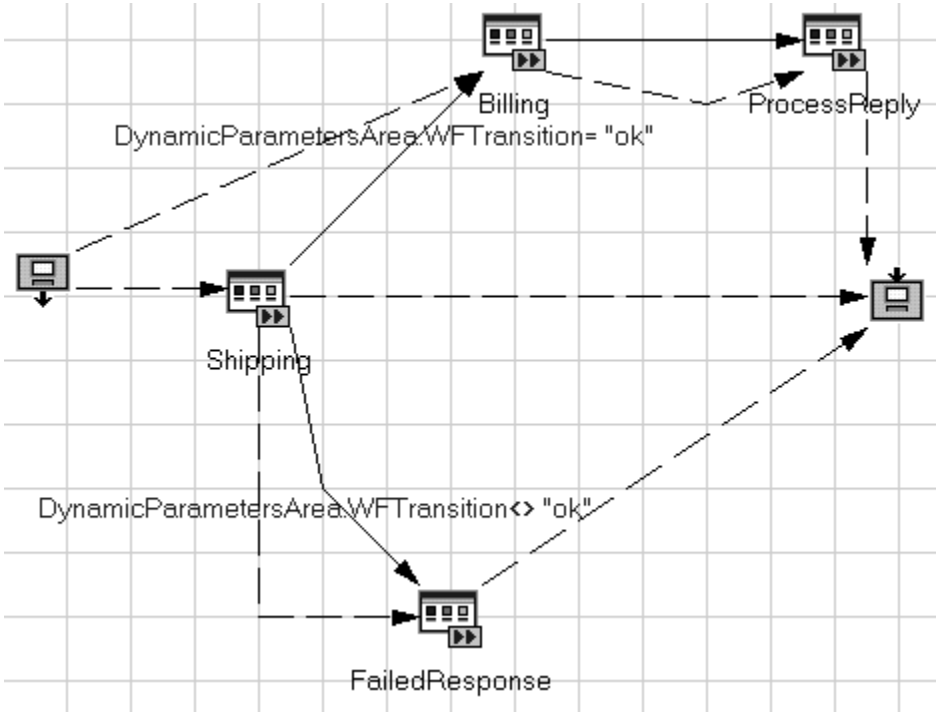


Figure 31: SyncTwoBackEndsIDM workflow process template.

Verbal explanation

The SyncTwoBackEndsIDM process uses information provided to the process to route an incoming message to the supplied destinationLogicalIds, in order. The response returned from the first destination is checked and routed to the next activity based on the value of the response message's cmdStatus. The original message is then routed to the second supplied destinationLogicalId. The second destination's response message is then routed back to the originating System. The process ends.

Recommended use

This workflow can be used by any messageType that conforms to the IBM IDM (that is, uses cmdStatus to indicate message response status) that requires the same message to be routed to two destinations serially.

TwoBackEnds workflow process template

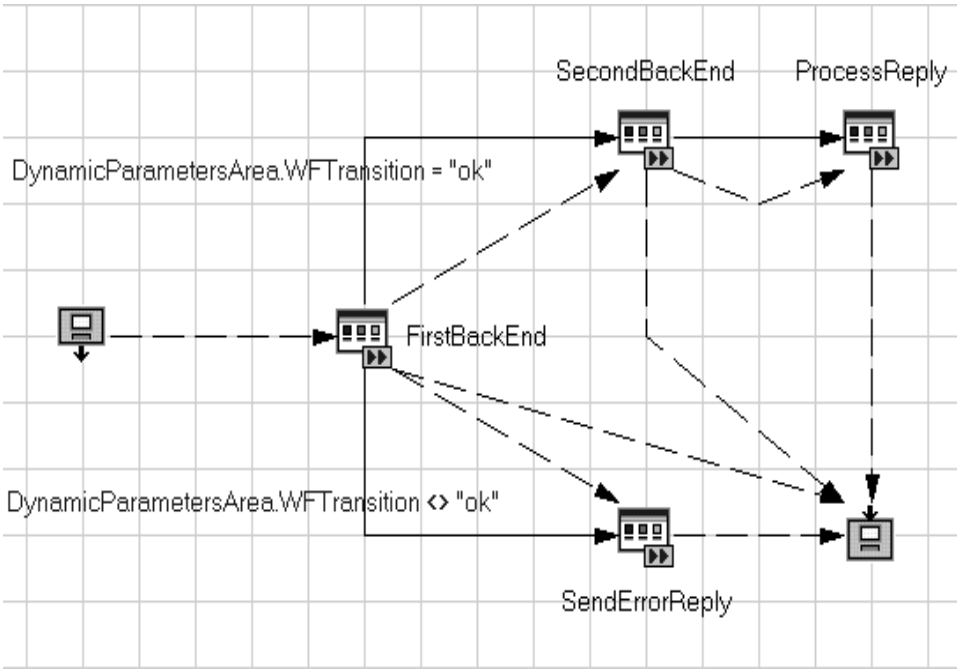


Figure 32: TwoBackEnds workflow process template.

Verbal explanation

The TwoBackEnds process uses information provided to the process to route an incoming message to the supplied destinationLogicalIds, in order. The response returned from the first destination is checked and routed to the next activity based on the value of the Message's cmdStatus. The response message from the first destination is then routed to the second supplied destinationLogicalId. The second destination's response message is then routed back to the originating System. The process then ends.

Recommended use

This workflow can be used by any messageType that conforms to the IBM IDM (that is, uses cmdStatus to indicate message response status) that requires a message to be routed to destination A with the response from A being routed on to B. The response from B is routed back to the originator.

PublishOnly workflow process template

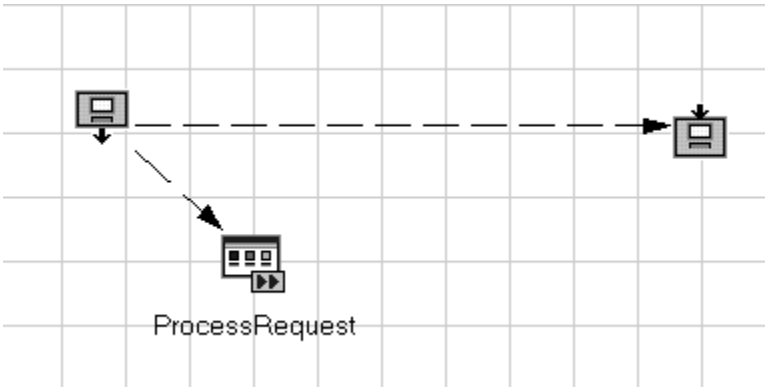


Figure 33: PublishOnly workflow process template.

Verbal explanation

The PublishOnly process uses information provided to the process to send an incoming message to WMQI Enabler for possible publishing to the supplied PublishTopic. The process ends. The message is not routed to any specific destinationLogicalId.

Recommended use

This workflow can be used by any messageType that conforms to the IBM IDM (that is, uses cmdStatus to indicate message response status) that wants a Publish performed.

The first three workflow processes described above are used by the OAG use case/test case suites. TwoBackEnds is utilized in the TwoBackEnd example. No shipped example of the PublishhOnly workflow process is provided.

Alternative to using MQSeries Workflow

WMQI Enabler is a pre-assembeld solution that combines MQSeries products in an integrated architecture and reference implementation. WMQI Enabler uses MQSeries Workflow for the following reasons:

- Restart/recovery.
- State management of the message flow.
- Mechanism for supporting long running transactions.

- Mechanism for supporting compensatory transactions.
- History and reporting capabilities.
- Task management infrastructure.
- Standardized (WFMC) API's for process integration.
- Scalability.
- Assured delivery of the request and response message pair.

WMQI Enabler uses the message profile to indicate how a message should be delivered.

The first delivery option is the "transactional message processing supervisor" implemented using MQSWF.

The second delivery option is the "lightweight message processing supervisor" (LMPS). This lightweight message processing supervisor is essentially the HUB_IN_Flow that processes the message through the WMQI Enabler features so that it can be directly delivered to the queues for the target adapter represented by the symbolic destination id, "DestinationLogicalId".

The advantage of the LMPS is that it allows the design to develop a highly optimized path to the target application, yet still provides access to specific WMQI Enabler functions for cross referencing keys, symbolic destination resolution, logging, system availability management,store and forward, and logging.

There are a number of design considerations when opting for the LMPS approach.

- LMPS is only capable of delivering content to a single target.
- LMPS uses an "optimistic processing model" that relies on a response from the target adapter in order to trigger a response to the source application. This means that unlike the transactional message processing supervisor, adapters have to be more sophisticated to handle the fact that the application has an error or exception.

Holosofx

The Holosofx company has created a tool that is complementary to the MQSeries Workflow tool and can be used for the development of workflow process templates. Holosofx Business Process Management (BPM) Suite© offers a solution for process management in today's business world. The BPM Workbench can be utilized to create process charts that are exportable to the MQSeries Workflow tool and, thus, is fully compatible with the WMQI Enabler product.

BPM components

The BPM Suite consists of three components:

- BPM Workbench
- BPM Monitor
- BPM Server

BPM Workbench further consists of a Business Modeler, UML Modeler, Xform Designer, and XML Mapper. With this workbench tool, it is possible to create flow chart based process modeling, perform cost and time analysis, create a process redesign report template, perform process simulation, do UML-based object modeling, create a GUI design, and create XML-based application communication. With this component, business processes can be modeled, design work can be reused, business models printed, and the results exported to the IBM MQSeries® Workflow tool. This tool provides a powerful analytical tool suitable for the professional operations analyst, and individuals working within the process are often the best candidates to model and analyze the process.

The BPM Monitor provides real time process tracking. One of its features is time, cost, and performance evaluation. In addition, a Web-based interface is available. It has as components both a "workflow dashboard" and a "business dashboard."

The BPM Server provides intra-company policy, procedure, and process distribution. It also has a Web based interface and is made up of a repository and has Web Publishing capability.

Potential use with WMQI Enabler

Holosofx was used to generate workflow process templates to validate its compatibility with WMQI Enabler. Holosofx BPM Workbench version 4.1 build 4205 was used to construct the workflows and it operated seamlessly with MQSeries Workflow version 3.3. The use of the tool requires the utilization of the IBM MQ Workflow edit mode for consistency with the MQSeries Workflow product. This mode is one of the eight editing modes offered within the Holosofx BPM Workbench product. The business process is defined within the BPM Workbench product through the use of the repositories: organization and process.

Within the organization repository, the domain, systems, queues, and applications are specified.

Within the process repository, the data is defined as "Phis" (input and output data) and the logic of the diagram can be specified through the use of decisions and choices.

The graphical interface for the process diagrams is based on this information and offers choices of task and process, similar to the workflow activities and processes, the "Phis" are separate icons for the data, the decisions and decision choices represent the logic of the diagram, and there is a connector for graphical representation of the business flow.

In addition, the MQSeries Workflow process templates can be imported and exported to the MQSeries Workflow tool by selecting **File** on the menu bar and clicking **Import/Export**. Here, several alternative choices are also presented for achieving this activity.

Within the BPM Workbench tool, numerous reports may be run to analyze costs, time expended, and performance. The process templates may be validated and the processes simulated prior to moving them into MQSeries Workflow. Using the BPM Monitor, these processes may be monitored in real time as well.

More information on Holosofx can be obtained at ***www.holosofx.com***.

Chapter 8
State tags

The usage of the state tags can vary from system to system due to the variances in the functionality that needs to be mimicked through WMQI Enabler. It is suggested that a clear understanding of the keys used for the front-end and back-end be clearly documented before beginning because the state tags represent an action that is (or was, depending on the status of the message) performed on the key. Further, the intent of the data, as described in the use case, and not necessarily the associated command, should be used as the primary factor in determining how the state tag values should be set for each aggregate within a message.

Example

An AddParty message will use "add" for all alternate ids, not because it is an add of party information. Adding a party generally entails that all aggregates and their data associated with the new party are actually new and need to be added.

Compare this with an "addClaim" message and you will see that the states do not necessarily correspond with the fact that this is an add command. In the case of an "addClaim", there are usually references to policy and coverage aggregates and their data. Since the policy and coverages aggregates exist as the basis for making the claim, the state tags associated with the aggregates for policy and coverage data, will use the tag value of "exists" so that the keys can be translated to their appropriate values between the source and target systems. The claim aggregate, on the other hand, will use “add”, since it is actually being created.

Compare this to a modification of a Policy. In most cases changes to existing data would require that the state tags be set to "exists" for the aggregates containing the data. This is because the aggregate exists and the keys need to be translated between the source and target applications so that they can properly reference the correct aggregate and make the required modifications to the appropriate attributes.

However it is also possible to modify a policy by adding new pieces of data to the policy. In these types of cases the state tag must be set to "add" for the affected aggregates.

It is necessary to look at the intent of the data contained in affected aggregates, as described by the use cases in order to determine how the value of the state tag should be set. Given this, there is still value in looking at types of messages and their corresponding usage of the state values in order to establish some general guidelines.

For a list of state definitions, see *Appendix C*.

An example of the KeyGroup section of the IAA-XML message:

```
<KeyGroup id="K1"
keyGroupType="InsurancePolicySearchCriteria">
<AlternateId sourceLogicalId="PolicySYS"
state="referenced"/>
```

state="exists"

For inquiry messages, the state attribute should be set to "exists" on the request portion of the message. As the back-end creates a message, the system will set the state to "referenced" within the response portion of the message. Once the response enters the hub, the hub will continue the translation of keys and change the state to "referenced" as it sends it back to the front-end. In addition, values are required in the KeyGroupType, state, value and sourceLogicalid tags.

state="add"

For add messages, it would be typical to use "add" as a state value. In our use cases we will use "referenced" on the request message to assist with the systems handling rollbacks. As the back-end creates their message, the system will set the state to "add" within the response portion of the message. Once the response enters the hub, the hub will create the UUID to associate all keys provided and will change the state to "added" as it is sent back to the front-end system. In addition, values are required in the KeyGroupType, state, value, and sourceLogicalid tags. Additionally, when another system wishes to attach keys to an existing UUID, the "add" state is also used. In this case either the UUID or an alternate id entry containing an existing value, system id, and state of "exists" must be provided as the first entry in the key group. This existing alternate id or the UUID will be used as the basis for attaching the alternate ids with the state of "add".

state="modify"

For modification messages where the intent is to modify the key stored in the cross-reference file we would use "modify" as a state value for the request portion of the message. CRF will change the key entry in the cross-reference file. If the front-end system doesn't change keys during a modification, then you should use the "exists" value. Once the hub returns the message to the front-end, it will change the state to "modified". In addition, values are required in the keyGroupType, state, value, sourceLogicalid, and new value tags.

For modification messages where the intent is to modify data in existing aggregates, the state tags should be set to "exists". In addition, values are required in the KeyGroupType, state, value, and sourceLogicalid.

In the event that the only change desired is a change to the attributeString, issue a "modify" where value and newValue are identical. Whatever is specified in the attrbiuteString will be updated on the CRF table.

state="delete"

For delete messages, the front-end system will set the state value to "delete" for the request. The back-end system will also set the response value to "delete" and both keys will be marked inactive in the CRF table. There will be a change in state to "deleted". In most scenarios, a system will not truly delete a record but may mark it as invalid; therefore we may only see "delete" on a request message without needing a response back from the back-end system. In addition, values are required in the KeyGroupType, state, value, and sourceLogicalid tags.

For query type messages, generally the request message doesn't require KeyGroup or state information, because we are asking the target application to find information based on search criteria instead of keys. (As a result, when the target application responds we generally do not need to use the cross reference function for the response. This is because the response message generally returns all of the items that match the search criteria and their keys.) Since the front-end does not have equivalent keys, there is no reason to translate the back-end keys. If the front-end decides to persist this information, it should use an add message to attach to the existing entries from the back-end system as described above.

For get and put messages, the state tag should be set to "exists" so that the get/put request key for the front-end can be translated into the corresponding equivalent in the back-end system.

Appendix A
State definitions

Action to be performed/that was performed on the key where:

referenced
Means the key is just information sent from one system to another.

add
Means the key needs to be added to CRF, only for message into hub.

added
Means the key was just created in the CRF, only for message coming from hub.

exists
Means the key is in the CRF & should be translated for another system specified by the destinationLogicalID.

modify
Means the key was modified and needs to be changed in the CRF, only for message into hub.

modified
Means the key was modified in the CRF, only for message coming from hub.

delete
Means the key needs to be removed from the CRF, only for message into hub.

deleted
Means the key was removed from the CRF, only for message coming out of the hub.

Appendix B
Subflow descriptions

AddWorkArea

Creates the WorkArea and populates it. The WorkArea provides an information wrapper that is used to pass parameters between MQSI nodes. The WorkArea may hold multiple messages. If the message does not contain Message or WfMessage as the high level tag, an error is generated. If the MQMD MsgType field has MQMT_REQUEST, a field in the work area is set to indicate that this is a synchronous message. Otherwise, it is identified as an asynchronous message.

AdvancedInput

Reads a message in from an MQ queue.

Passes control to the try path of a catch block. If an error occurs during the execution of the try path, flow control is throw back to the catch path.

Try

Executes the AddWorkArea subflow.

Executes the LogMessage subflow.

Catch

Executes the LogOriginalMessage subflow.

Executes the CauseDataRollback subflow.

AdvancedOutput

Executes the LogMessage subflow.

Executes the BuildDestinationList subflow.

Executes the RemoveWorkArea subflow.

Writes the message to the specified Queue.

BuildDestinationList

Clears out the destination list.

If there are items in the execution list

then

Sets up the defaults of the destination list

Defaults are :

Name = 'Defaults';

transactionMode = 'automatic';

persistenceMode = 'automatic';

newMsgId = 'no';

newCorrelId = 'no';

segmentationAllowed = 'no';

alternateUserAuthority = 'no';

Moves every item in the results list into the historylist.

Moves the reply to information in work area into the mqmd. Nulls out the mqmd first, to insure the new mqmd information is used.

Takes every item in the execution list and moves it into the the MQDestinationList data.

CheckOfflineBodyCategory

Validates the incoming HUBONLYOFFLINE request message contains a valid body category and routes the message to the corresponding label in the HUB_ONLY_OFFLINE flow.

CheckOnlineBodyCategory

Validates the incoming HUBONLYONLINE request message contains a valid body category and routes the message to the corresponding label in the HUB_ONLY_ONLINE flow.

CRF

Creates a working copy of the current message for use by the CRF subflows.

For every CrfActionGroup in a message, loop through the key groups and process them.

If an error occurs during CRF processing, return the original message to the requester instead of the "in progress" working copy.

The symbolic of each key group is verified through the SDRFunction sub flow. If an error occurs in that subflow, CRF captures it and processes it.

ErrorHandler

Call the ProcessNLSError subflow to build the NLS error message.

Sets the destination list to look for the appropriate route to label.

Sets an error indicator for use by the KillProcess subflow.

Routes the error via a RouteTo node with mode of Route To Last.

FormatHubResponse

Formats Hub command responses. There is a node with promoted attributes that allows for unique code to be generated for each command, such as setting a status flag.

An additional node is used for the common functions of building a response, such as setting the destination of the message, populating the SDR execution list, and formatting the response message.

GetINSEntry

Get the active INS entry out of the Install_Data_Table in the FSE_SYP database. This table provides the installation information on this system such as the platform, version, and default language.

Calls FormatHubResponse to generate the response message to the requesting system.

GetMessageProfile

Gets the message profile for this body category from the Message_Profile_Table on the FSE_MSGP database. If the body category does not exist, this is a Hub Only message and the cmdType is used.

Moves the temporary sequence flag field into the used sequence flag field.

Gets the system interaction list for this body category from the System_Interaction_Table.

If a valid message profile does not exist for this message type, or the message is disabled from hub processing with a MQSIMessageEnabledFlag of False, an error is generated.

GetNLSEntry

Gets the error message information for a specified error number.

Calls FormatHubResponse to generate the response message to the requesting system.

GetSDREntry

Gets the Queue Manager for this system symbolic from the SDR_Table on the FSE_SDR database.

Gets the Queue for this system symbolic from the SDR_Table.

Executes the FormatHubResponse subflow.

GetSystemProfile

Gets the system back up list for this system symbolic from the System_Backup_Table on the FSE_SYSP database.

Gets the store forward flag list for this system symbolic from the System_Store_Flag_Table.

Gets the values of all flags and fields, including language, shutdown information, active flag, return codes, and error message destination from the System_Status_Table.

Executes the FormatHubResponse subflow.

HUB_IN_LDAP

Duplicates HUB_IN, but uses SDR implemented with an LDAP interface.

HubOnlyMessageRouter

Based on the flag set in the node PrepHubOnlyFlag, sends the message to HUB_ONLY_ONLINE, or HUB_ONLY_OFFLINE message queues. The flag is 0 for not Hub Only, 1 for Hub Only Offline, and 2 for Hub Only Online.

If the message is not a Hub Only command, it continues through the out terminal.

KillProcess

Updates the Process_State_Table indicating that the process is complete or had an error.

If this was the last process using the current system, sets the Session_Table indicating it is no longer active.

Each system using the process that requested a shut down, is now allowed to shutdown if they:

Do not have a process currently in system interaction check.

Do not have any active sessions.

Send a message to each of these systems indicating that it can now shut down.

Process the publishing of a message by calling the PluggablePublish subflow.

KillSession

Deactivates the record for this session in the Session_Table on the FSE_SESS database.

Gets all the process ids using this session id from the Process_State_Table.

For each process in the session:

Sets up to kill the process.

Executes the KillProcess subflow.

LogError

Sets up the process id, getting the information from the message or wfmessage in the WorkArea structure.

Sets up the session id and message, getting the information from the message or wfmessage WorkArea structure.

Inserts the error into the Error_Table on the FSE_ERRL database.

Inserts the message into the Message_Table.

If exceptions exists, inserts them, one at a time, into the Exception_Table.

Stores trace information into the Trace_Table and the Trace_Table, as appropriate.

Checks the current error to ensure it is not the same as the last error logged,

If they are identical, then

The process is looping. Break out of the loop.

Creates an error response, adding in workflow information if the error is a workflow error.

Executes the SDR subflow.

Sets up the destination list and executes the AdvancedOutput subflow.

Executes the KillProcess subflow and starts a database rollback.

LogErrorOnly

Processes only the database update found in the LogError subflow.

LogMessage

For messages with a non-NULL msgID,

Generate process ID or assign an existing process ID if correct information is populated.

While there are items in the results list, move them to the history list and then clear them from results.

While there are items in the ExecutionList, insert each messageItem message to the Message_Log_Table on the FSE_MSGL database.

LogoffAndRespond

Gets the active processes flag of the session from the Session_Table on the FSE_SESS

If the active processes flag does not exist, then

Sets an invalid session indication.

Else

Checks to see if the session has active processes.

If there are active processes, then

deny logoff.

Else

Updates the session table to invalidate session id.

Executes the FormatHubResponse subflow.

LogonAndRespond

Checks the authentication id passed in the message, to see if it exists in the Session_Authentication_Table on the FSE_SESS database.

If the authentication id is found, then

creates a session id and moves that id to the message.

Else

clears out the session id information.

Inserts the newly created session id into the Session_Table on the FSE_SESS database.

Executes the FormatHubResponse subflow.

LogOriginalMessage

Inserts a Message into the Original_Message_Table on the FSE_ERRL database

ProcessNLSError

First, the exception that is passed in is checked to see if it is a terminal error, such as a parsing error. If it is, the error is logged and processing finishes.

If this is not a terminal error, the exception is converted into NLS format.

Based on the language and error number, the error message is retrieved from the NLS error database and formatted.

The error is passed to LogError for processing.

ProcessSequenceValidation

If sequence validation is required by the message profile then do the following:

Gets the timestamp from the last process that completed on this session.

Uses the timestamp to get the message type name of that completed message from the Process_State_Table.

Compares the message type name to the message dependency of our current message.

If the dependency is met, then

Transfer control to the valid terminal.

Else,

Executes the SequenceInvalid node.

ProcessSession

If session validation is required by the message profile do the following:

If true,

Executes the ProcessSessionValidation subflow.

Executes the ProcessSequenceValidation subflow.

Transfers control to the out terminal for every path through the subflow.

ProcessSessionValidation

Checks for a valid session.

If valid, then

Updates the session record information.

else

Generates an error message.

ProcessSystemInteraction

If system interaction is enabled,

Executes the ProcessSystemInteractionCheck subflow.

If there are Interaction Problems, executes the StoreMessage subflow.

ProcessSystemInteractionCheck

Updates the process state to indicate system interaction check is in progress.

Gets the time stamp at the start of the system interaction check from the Process_State_Table on the FSE_SESS database.

Checks for required systems. If a system is not there, checks for an available backup. If required system and all of its backups are unavailable, sets the InteractionProblemsFlag.

If checking multiple systems and one required system is not there, none of the systems receive the message. All required systems must be up. If a required system is not up, an available backup must be found up. Otherwise, an error is created.

Updates process state to indicate system interaction check is complete.

If system interaction problems were found, transfer control to the problems terminal. Else transfer control to the out terminal.

ProcessWorkflowRequest

If a response is required to workflow,

Checks that correlId, message id, original id, and work flow data structure name are included in the Request.

If any is NULL, error and exit.

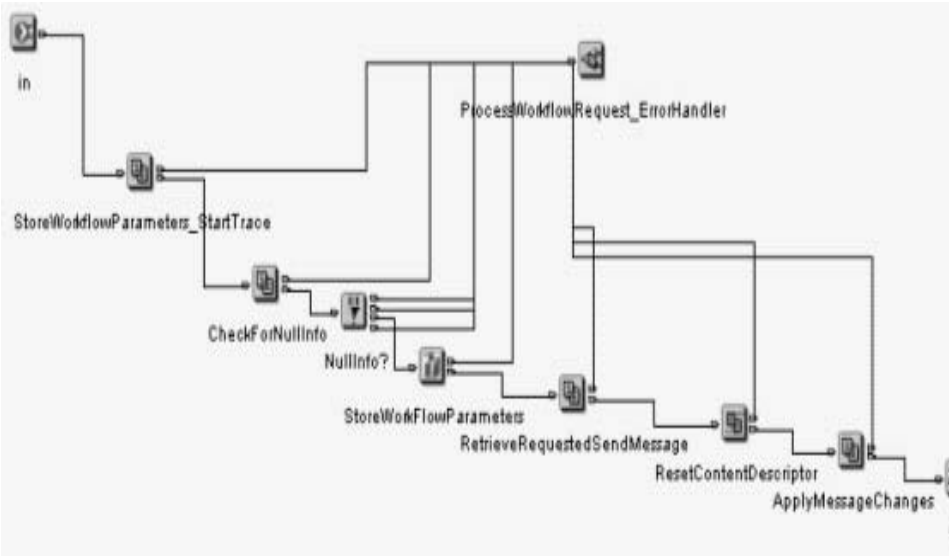
Stores the work flow parameters into the Workflow_Parameters_Table on the FSE_WFCO database.

Retrieves the message stored to the MESSAGE_LOG_TABLE on the FSE_MSGL database to the WorkaArea.

Applies any workflow requested messages changes.

Modification for Complex Business Processes Support (UFCU)

Once a Workflow process generates a message for communication with WMQI Enabler, the message is sent to MQWF_OUT. The message flows to ProcessWorkflowRequest_Subflow.



- Database Node StoreWorkflowParameters stores the reply to information specified in the MQMD header in the Workflow_Correl_Table, along with the message id, workflow correl id, message name, original message id, and the data structure name that the workflow expects to see in its response message. The reply to information is used in HUB_RWF_IN to send the response back to the requesting instance of Workflow.

- Compute Node RetrieveRequestedSendMessage examines the incoming Workflow message for the MessageTemplateFlag. If this flag is “True”, the data in the MessageName tag is used to pull a template out of the Message_Template_Table.

  If the flag is not “True”, processing continues as before, using MessageName and process id to pull a requesting message from the Message_Log_Table

- After the retrieved message is reset in the ResetContentDescriptor Node, Compute Node ApplyMessageChanges adds tag TemplateId to the Message. This tag is a temporary field that will be used to identify necessary correlation information to HUB_IN. It contains the message id within the MQMD header of the requesting WorkflowMessage.

- The message then flows back to MQWF_OUT. Within Compute Node PrepSDRParameters, if this message is a template, then queue HUB_IN on queue manager MQSIQM is set as the destination for the message.

Within Compute Node PrepBuildDestinationList, as with all messages in this flow, queue HUB_RWF_IN is specified as the response queue.

The message template is then sent to HUB_IN with a new message id in the MQMD header, and it is treated as all messages are.

Within Compute Node ProcessWorkFlowCheck of ProcessWorkFlowStart, if the TemplateId exists within the message and workflow processing is required, it is used to set the OriginalMessageId in the workflow message being created. If workflow processing is not required the TemplateId is used to replace the MQMD header MsgId and the message is sent to whatever backend is specified in its message profile or in its destinationLogicalId.

ProcessWorkflowResponse

Loads static work flow related information from the Message_Profile_Table on the FSE_MSGP database

Moves the temporary field of sequence validation to MQSIMessageSequenceValidationFlag.

Gets the system interaction list for this message from the System_Interaction_Table.

Gets information stored to the WorkFlow_Correl_Table on the FSE_WFCO database for the response message.

Loads the Workflow container header information.

Loads the Workflow container CommonArea.

Loads the Workflow container SystemInfoArea with the system interaction list.

Loads the Workflow container DynamicParametersArea.

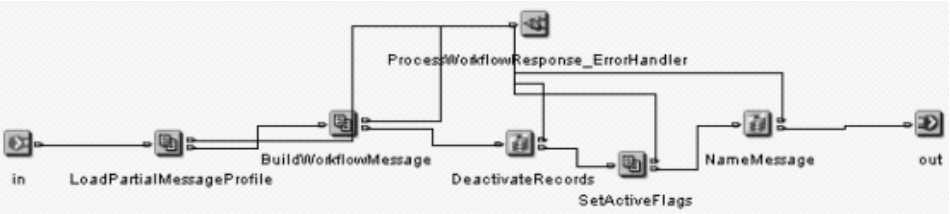
Sets up the destination list parameters.

Sets the active flags for the system in the interaction list from the Session_Processes_And_System_Usage_Table on the FSE_SESS database.

For our current message id, updates the message log with a message name requested by work flow.

Modification for Complex Business Processes Support (UFCU)

Once the message template processes to completion, the response message is sent to HUB_RWFIN. This process is controlled from MQWF_OUT; when the template is sent to HUB_IN, HUB_RWF_IN is specified as the requesting system reply to information.



ProcessWorkflowResponse_Subflow within HUB_RWF_IN builds the message that responds to the Workflow process that invoked the template. Within Compute Node BuildWorkflowMessage, the correl id in the MQMD header is used to pull the appropriate information out of the Workflow_Correl_Table. At this point the correl id is populated with the message id of the original Workflow message. This is controlled in one of two ways.

If the template is processed through Workflow, the message id is saved in the OriginalMessageId field within the workflow process. When it is last processed in MQWF_OUT, this OriginalMessageId is moved into the correl id of the MQMD header.

If the template is not processed through Workflow, the message id of the original Workflow message replaces the message id of the requesting template message. Once the template message reaches the back end system, the message id is moved to the correl id of the MQMD header.

The Workflow correlation information is pulled from the Workflow_Correl_Table and used to build a response message to Workflow, and the initial process within workflow is completed.

ProcessWorkflowStart

If the message profile indicates that WorkFlow Management is enabled,

then

Gets the WorkFlow parameters list from the Workflow_Parameters_Table on the FSE_MSGP database.

Loads the Workflow container CommonArea.

Loads the Workflow container SystemInfoArea with the system interaction list.

Loads the Workflow container DynamicParametersArea.

Sets up the destination list parameters.

Transfers control to the workflow terminal.

Else

Sets the CRF offset to the last message in the list of messages.

Transfers control to the no workflow terminal.

Publish

Adds the correct MQRFH2 parameters and a publishable message to the outgoing message. Sends this extract to a Publish node. Messages will be published to a topic specified in the message profile.

PluggablePublish

Provides a publish function that can be used anywhere within the flows where the MQSIWorkArea still exists within the message. It sets up the publish topic based on the message profile contents and checks to see if publishing is required based on flags in the message profile, flags in the message, and whether or not it is an error situation.

RemoveWorkArea

Passes the output destination list through to the output node.

Removes the WorkArea.

Moves the XML version, encoding and standalone parameters stored back into the outgoing message.

SDR

Calls the SDRFunction subflow, and captures it's errors.

SDRFunction

Moves every item in the results list into the history list.

Clears out the SDR error tag.

For every item in the execution list.

Checks to see if the item in the execution list is in the historylist.

If the item is in the historylist, move it to the results list. Otherwise, get result from the SDR_Table on the FSE_SDR database. Validates SDR results for correct Queue and QueueManager.

Clears the execution list.

If an error was found in the SDR processing, transfer control to the ValidatSDR?_False terminal.

SDR_IN_LDAP

Calls the SDRFunction_LDAP subflow, and captures it's errors

SDRFunction_LDAP

Moves every item in the results list into the history list.

Clears out the SDR error tag.

For every item in the execution list.

Checks to see if the item in the execution list is in the historylist.

If the item is in the historylist, move it to the results list. Otherwise, get result from the SDR Structure on a specified LDAP server. Validates SDR results for correct Queue and QueueManager.

Clears the execution list.

If an error was found in the SDR processing, transfer control to the ValidatSDR?_False terminal.

StoreMessage

If Store Message is required,

Extract the BLOB portion of the WorkArea message and store it to the Stored_Message_Table on the FSE_STOF database.

Checks the system interaction requirements for this message, and stores correlating information of the stored message into the Interaction_Dependency_Table.

Updates the Process_State_Table, indicating the message was stored for forwarding when the system is restarting.

Else

Pass on the exception generated in ProcessSystemInteractionCheck to the Error Handler.

SetSubscription

Generates subscription information based on the COMMAND section of the message.

Sends that information to the Broker's control queue.

Builds response message for the requesting system.

Used to register and drop subscriptions.

SystemRestart

Deactivates the existing records of the System_Status_Table on the FSE_SYSP database for the system symbolic.

Inserts a new record into the System_Status_Table for our system symbolic.

Checks the Interaction_Dependency_Table on the FSE_STOF database for stored messages stored with our current system as a destination.

For every stored message, retrieve the original message from the Stored_Message_Table and re-parses the message.

Once all stored messages and restarted, executes the FormatHubResponse subflow.

SystemShutdown

Sets the requested shutdown flag in the System_Status_Table on the FSE_SYSP dB to true for the system symbolic.

Sets the requested shutdown flag in the Session_Processes_And_System_Usage_Table on the FSE_SESS db to true for the system symbolic.

Gets the requested shutdown timestamp from the System_Status_Table for every system with our symbolic that wants to shutdown.

Checks the Process_State_Table to see if any of the systems that want to shutdown are currently in system interaction check state. If a system that wants to shutdown is in system interaction check or has a process currently using it, setup to deny shutdown.

If the system cannot shutdown, set the blocked by system interaction flag in the System_Status_Table to true.

If the system can shutdown, sets the Active flag in the System_Status_Table to false.

Executes the FormatHubResponse subflow.

TraceLog

If the trace flag is not in the work area, retrieve it from the message profile database

If trace is enabled, copies the information in the WorkArea TraceArea to the Trace_Table, then clears the WorkArea TraceArea.

UpdateINSEntry

Deactivates the active record in the Install_Data_Table within the FSE_SYSP database.

Inserts a new record in the table with the information in the COMMAND section of the message.

UpdateMessageProfile

Deactivates the records in the Message_Profile_Table with the current Message Type Name.

Deactivates the records in the System_Interaction_Table with the current Message Type Name.

Inserts a new record in the Message_Profile_Table with the information from the COMMAND section.

Deactivates the Workflow Parameters in the Workflow_Parameters_table for the current message type.

Inserts Workflow Parameters in the Workflow_Parameters_Table on FSE_MSGP with the information from the COMMAND section.

Deactivates entries on the System_Interaction_Table for the current Message Type and System Symbolic.

Inserts System_Interaction_Table record(s) with the information from the COMMAND section.

Executes the FormatHubResponse subflow.

UpdateNLSEntry

Deactivates all records for the error number specified in the message in the NLS_Error_Message_Table.

Inserts new records in the table with the information in the COMMAND section of the message.

UpdateSDREntry

Deactivates all SDR records in the SDR_Table with the current system symbolic.

Inserts a new record into the SDR_Table using the information passed in the COMMAND section.

Executes the FormatHubResponse subflow.

UpdateSystemProfile

Deactivates all existing records in the System_Status_Table with the current system symbolic.

Deactivates all existing records in the System_Backup_Table with the current system symbolic.

Deactivates all existing records in the System_Store_Flag_Table with the current system symbolic.

Inserts the supplied Backups into the System_Backup_Table.

Inserts the supplied Store Forward flags into the Store_Forward_Flag_Table.

Inserts the supplied System information into the System_Status_Table.

Executes the FormatHubResponse subflow.

Version1.2.2

Contains the version indicator 1.2.2.

Appendix C
WMQI Enabler routing diagram

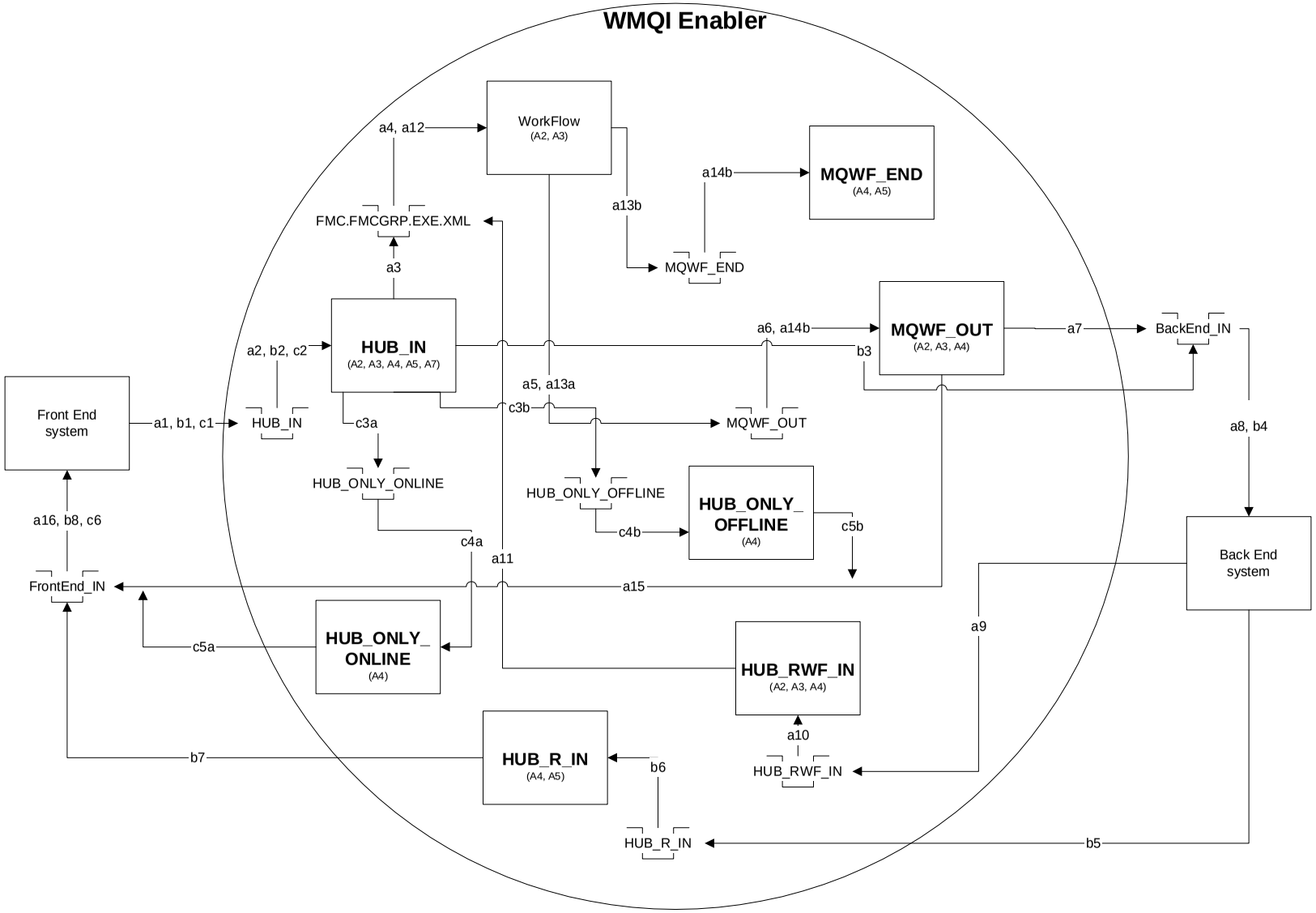
The illustration on the next page shows an WMQI Enabler routing diagram that demonstrates three possible scenarios.

Scenario a: Shows a message passing through the HUB that uses MQSeries Workflow. See steps a1 through a16 to follow its path. Steps a13a, a13b, a14a, and a14b are steps that run in parallel.

Scenario b: Shows a message passing through the HUB that does not use MQSeries Workflow. See steps b1 through b8 to follow its path.

Scenario c: Shows a hub-only message. See steps c1 through c6 to follow its path. Steps c3a, c4a, and c5a follow the HUB_ONLY_ONLINE path. Steps c3b, c4b, and c5b follow the HUB_ONLY_OFFLINE path.

In this diagram FMC.FMCGRP.EXE.XML is the standard input queue for Workflow.



Appendix D
MQSI WorkArea DTD

The MQSI_WorkArea DTD is as follows:

```
<!ELEMENT MessageGroup  (OriginalMessageOffset? , MessageItem* ,
Type? , ProcessId? , MessageName? , Message? , WFMessage?,
NLSErrorMesssage?, NLS_Results?, INS_Results?,
CompletedNLSErrorMessage? )>
<!ELEMENT OriginalMessageOffset  (#PCDATA )>
<!ELEMENT MessageItem  (Type? , ProcessId? , WfMessage? ,
MessageName? , Message? )>
<!ELEMENT Type  (#PCDATA )>
<!ELEMENT ProcessId  (#PCDATA )>
<!ELEMENT MessageName  (#PCDATA )>
<!ELEMENT Message  (#PCDATA )>
<!ATTLIST Message  sourceLogicalId CDATA  #IMPLIED >
<!ELEMENT WFMessage  (#PCDATA )>
<!ELEMENT Parameters  (SDR? , BuildDestinationList? , RemoveWorkArea?
, LogMessage?)>
<!ELEMENT SDR  (ExecutionList , ResultsList , HistoryList )>
<!ELEMENT MQSI_WorkArea  (MessageGroup? , Parameters? , ErrorList? ,
TraceArea? , MessageProfile? , Session? , SystemProfileArea? ,
CRF_WorkArea?, Publish?)>
<!ELEMENT ErrorList  (Error* )>
<!ELEMENT TraceArea  (StartTraceList? , EndTraceList? )>
<!ELEMENT MessageProfile  (MessageTypeName? ,
MQSISessionValidationFlag? , MQSIMessageSequenceValidationFlag? ,
MQSISystemInteractionCheckFlag? , WorkFlowManagementFlag? ,
WorkFlowQueueManager? , WorkFlowQueue? ,
WorkFlowReplyToQueueManager? , WorkFlowReplyToQueue? ,
WorkFlowSymbolic? , WorkFlowDataStructureName? ,
DefaultDestinationSymbolic? , MessageTypeDependency? ,
WorkFlowParametersList? , SystemInteractionList?,TraceFlag? ,
PublishFlag? , OverrideFlag? , PublishErrorsFlag? ,
MQSIMessageEnabledFlag?, WorkFlowProcessName?)>
<!ELEMENT Session  (SessionId? , SessionState? )>
<!ELEMENT SystemProfileArea  (SourceSystemProfile? , SystemSymbolic?
, StoreFlag? , DestinationSystemProfileList? , Language? )>
<!ELEMENT CRF_WorkArea  (MessageOffset )>
```

```
<!ELEMENT BuildDestinationList  (ExecutionList? , ResultsList? ,
HistoryList? )>
<!ELEMENT INS_Results (Hardware+platform? Default_Language?,
Prodcut_Version?)>
<!ELEMENT NLSErrorMessage (Messsage_Number?, Section?, NodeName?)>
<!ELEMENT NLS_Results (LanguageNumber?, MessageNumber?, Section?)>
<!ELEMENT Publish (PublishMessageOffset? , TopicList? )>
<!ELEMENT TopicList (Topic? )>
<!ELEMENT RemoveWorkArea  (OutGoingMessageOffset? )>
<!ELEMENT LogMessage  (ExecutionList? , ResultsList? , HistoryList?
)>
<!ELEMENT ExecutionList  (Item? , Destination? , QueueManager? ,
ReplyTo? )>
<!ELEMENT ResultsList  (Item? )>
<!ELEMENT HistoryList  (NumberOfItems? , Item* )>
<!ELEMENT Item  (SystemSymbolic? , Status? , QueueManager? , Queue?
, LogPointName? , LogMessageOffset? , LogPath? , Name? , Result? ,
ParameterName? , ParameterPath? , DefaultValue? , Destination? ,
RequiredParameterFlag? , ReplyTo? , RequiredInteractionFlag? ,
SystemMirror? , SystemBackup? )>
<!ELEMENT Section (Section_Number?, Value?, SectionNumber?,
MessageText?, SpaceBefore?, SpaceAfter?, TextOnly?)>
<!ELEMENT Section_Number (#PCDATA )>
<!ELEMENT Value  (#PCDATA )>
<!ELEMENT NodeName  (#PCDATA )>
<!ELEMENT Message_Number  (#PCDATA )>
<!ELEMENT MessageNumber  (#PCDATA )>
<!ELEMENT SectionNumber  (#PCDATA )>
<!ELEMENT MessageText  (#PCDATA )>
<!ELEMENT TextOnly  (#PCDATA )>
<!ELEMENT CompletedNLSErrorMessage  (#PCDATA )>
<!ELEMENT LanguageNumber  (#PCDATA )>
<!ELEMENT Hardware_Platform (#PCDATA )>
<!ELEMENT Product_Version (#PCDATA )>
<!ELEMENT Default_Language (#PCDATA )>
<!ELEMENT SpaceBefore  (#PCDATA )>
<!ELEMENT SpaceAfter  (#PCDATA )>
<!ELEMENT SystemSymbolic  (#PCDATA )>
<!ELEMENT Status  (#PCDATA )>
<!ELEMENT QueueManager  (#PCDATA )>
<!ELEMENT Queue  (#PCDATA )>
<!ELEMENT LogPointName  (#PCDATA )>
<!ELEMENT LogMessageOffset  (#PCDATA )>
<!ELEMENT LogPath  (#PCDATA )>
```

```
<!ELEMENT NumberOfItems  (#PCDATA )*>
<!ELEMENT OutGoingMessageOffset  (#PCDATA )>
<!ELEMENT Error  (#PCDATA )>
<!ELEMENT StartTraceList  (Item* )>
<!ELEMENT EndTraceList  (Item* )>
<!ELEMENT Name  (#PCDATA )>
<!ELEMENT Result  (#PCDATA )>
<!ELEMENT ParameterName  (#PCDATA )>
<!ELEMENT ParameterPath  (#PCDATA )>
<!ELEMENT DefaultValue  (#PCDATA )>
<!ELEMENT RequiredParameterFlag  (#PCDATA )>
<!ELEMENT MessageTypeName  (#PCDATA )>
<!ELEMENT MQSISessionValidationFlag  (#PCDATA )>
<!ELEMENT MQSIMessageSequenceValidationFlag  (#PCDATA )>
<!ELEMENT MQSISystemInteractionCheckFlag  (#PCDATA )>
<!ELEMENT WorkFlowManagementFlag  (#PCDATA )>
<!ELEMENT WorkFlowProcessName (#PCDATA )>
<!ELEMENT WorkFlowQueueManager  (#PCDATA )>
<!ELEMENT WorkFlowQueue  (#PCDATA )>
<!ELEMENT WorkFlowReplyToQueueManager  (#PCDATA )>
<!ELEMENT WorkFlowReplyToQueue  (#PCDATA )>
<!ELEMENT WorkFlowSymbolic  (#PCDATA )>
<!ELEMENT DefaultDestinationSymbolic  (#PCDATA )>
<!ELEMENT MessageTypeDependency  (#PCDATA )>
<!ELEMENT PublishTopic (#PCDATA )>
<!ELEMENT PublishFlag (#PCDATA )>
<!ELEMENT OverrideFlag (#PCDATA )>
<!ELEMENT PublishErrorsFlag (#PCDATA )>
<!ELEMENT MQSIMessageEnabledFlag (#PCDATA )>
<!ELEMENT WorkFlowParametersList  (Item? )>
<!ELEMENT SystemInteractionList  (InteractionProblemsFlag? , Item? )>
<!ELEMENT sourceLogicalId  (#PCDATA )>
<!ELEMENT Destination  (QueueManager? , Queue? )>
<!ELEMENT ReplyTo  (QueueManager? , Queue? )>
<!ELEMENT InteractionProblemsFlag  (#PCDATA )>
<!ELEMENT SessionId  (#PCDATA )>
<!ELEMENT SessionState  (#PCDATA )>
<!ELEMENT SourceSystemProfile  (SystemSymbolic? , StoreFlag? )>
<!ELEMENT StoreFlag  (#PCDATA )>
<!ELEMENT DestinationSystemProfileList  (#PCDATA )>
<!ELEMENT MessageOffset  (#PCDATA )>
<!ELEMENT RequiredInteractionFlag  (#PCDATA )>
<!ELEMENT SystemMirror  (#PCDATA )>
```

```
<!ELEMENT WorkFlowDataStructureName  (#PCDATA )>
<!ELEMENT SystemBackup  (#PCDATA )>
<!ELEMENT WfMessage  (#PCDATA )>
```

Appendix E
Example MQSI WorkArea

An example MQSI_WorkArea structure is as follows:

```
<MQSI_WorkArea>
   <MessageGroup>
      <OriginalMessageOffset>1</OriginalMessageOffset>
      <MessageItem>
          <Type>IAA-XML</Type>
         <ProcessId></ProcessId>
         <MessageName></MessageName>
         <Message>
         </Message>
      </MessageItem>
      <MessageItem>
         <Type>WorkFlow</Type>
         <ProcessId></ProcessId>
         <WfMessage>
         </WfMessage>
      </MessageItem>
      <MessageItem>
         <Type>IAA-XML</Type>
         <ProcessId></ProcessId>
         <Message sourceLogicalId="Siebel">
         </Message>
      </MessageItem>
      <NLSErrorMessage>
         <NodeName>aComputeNode</NodeName>
         <Message_Number>1499</Message_Number>
         <Section>
            <Section_Number>1</Section_Number>
            <Value>Something bad happened</Value>
         </Section>
      </NLSErrorMessage>
      <NLS_Results>
         <LanguageNumber>10</LanguageNumber>
         <MessageNumber>N101499</MessageNumber>
          <Section>
```

```
              <SectionNumber>1</SectionNumber>
              <MessageText>Something bad happened</MessageText>
              <SpaceBefore>True</SpaceBefore>
              <SpaceAfter>True</SpaceAfter>
              <TextOnly>True</TextOnly>
          </Section>
</NLS_Results>
      <INS_Results>
         <Hardware_Platform>NT</Hardware_Platform>
         <Product_Version>01</Product_Version>
         <Default_Language>EN_US</Default_Language>
      </INS_Results>
      <CompletedNLSErrorMessage>N101499 Something bad happened
</CompletedNLSErrorMessage>
   </MessageGroup>
<Parameters>
      <SDR>
         <ExecutionList>
            <Item>
               <SystemSymbolic>Siebel</SystemSymbolic>
            <Item>
         </ExecutionList>
         <ResultsList>
            <Item>
               <SystemSymbolic>CIIS</SystemSymbolic>
               <Status>Processed Successfully</Status>
               <QueueManager>CIISQM1</QueueManager>
               <Queue>CIISIN1</Queue>
            </Item>
         </ResultsList>
         <HistoryList>
            <Item>
               <SystemSymbolic>SLU</SystemSymbolic>
               <Status>Processed Successfully</Status>
               <QueueManager>SLUQM1</QueueManager>
               <Queue>SLUIN1</Queue>
            </Item>
         </HistoryList>
        </SDR>
```

```
 <BuildDestinationList>
         <ExecutionList>
               <ReplyTo>
                   <QueueManager>MQSIQM</QueueManager>
                   <Queue>MQWFEND</Queue>
               </ReplyTo>
               <Item>
                  <Destination>
                     <QueueManager>FMCQM</QueueManager>
                     <Queue>FMC.FMCGRP.EXE.XML</Queue>
                  </Destination
               </Item>
         </ExecutionList>
         <ResultsList>
              <ReplyTo>
                  <QueueManager></QueueManager
                  <Queue></Queue>
<\ReplyTo>
               <Item>
                  <Destination>
                     <QueueManager>QM5</QueueManager>
                     <Queue></Queue>
                  </Destination>
               <Status>Error: Destination.Queue is NULL</Status>
            </Item>
         </ResultsList>
         <HistoryList>
            <Item>
            </Item>
         </HistoryList>
      </BuildDestinationList>
      <RemoveWorkArea>
         <OutGoingMessageOffset>1</OutGoingMessageOffset>
      </RemoveWorkArea>
      <LogMessage>
         <ExecutionList>
            <Item>
               <LogPointName>HUB_IN Log At Start</LogPointName>
               <LogMessageOffset>1</LogMessageOffset>
               <LogPath>
```

```
MQSI_WorkArea.MessageGroup.MessageItem[MQSI_WorkArea.MessageGroup.O
riginalMessageOffset].Message
                </LogPath>
             </Item>
          </ExecutionList>
          <ResultsList>
             <Item>
                <LogPointName>HUB_IN Log At Start</LogPointName>
                <LogMessageOffset>1</LogMessageOffset>
       <LogPath>
MQSI_WorkArea.MessageGroup.MessageItem[MQSI_WorkArea.MessageGroup.O
riginalMessageOffset].Message
                </LogPath>
                <Status>Processed Successfully</Status>
             </Item>
          </ResultsList>
          <HistoryList>
             <Item>
             </Item>
             <Item>
             </Item>
          </HistoryList>
       </LogMessage>
  </Parameters>
  <Publish>
       <PublishMessageOffset></PublishMessageOffset>
       <TopicList>
           <Topic>AddParty</Topic>
       </TopicList>
  </Publish>
  <ErrorList>
       <Error>LogMessage_Failure</Error>
       <Error>SDR_Failure</Error>
       <Error>SDR_Failure</Error>
  </ErrorList>
  <TraceArea>
```

```
   <StartTraceList>
           <Item>
              <Name>Check for duplicate Message names</Name>
              <Status>Incomplete</Status>
           </Item>
           <Item>
              <Name>Update database record to name message</Name>
              <Status>Incomplete</Status>
           </Item>
        </StartTraceList>
        <EndTraceList>
           <Item>
             <Name>Load WorkFlow correl parameters from database</Name>
              <Status>Processed Successfully</Status>
              <Result>WorkFlow correl data loaded</Result>
           </Item>
           <Item>
              <Name>Is this a WorkFlow message?</Name>
              <Status>Processed Successfully</Status>
              <Result>It is a WorkFlow message</Result>
           </Item>
        </EndTraceList>
  </TraceArea>
  <MessageProfile>
        <MessageTypeName>AddParty</MessageTypeName>
        <MQSISessionValidationFlag>True</MQSISessionValidationFlag>
  <MQSIMessageSequenceValidationFlag>False</MQSIMessageSequenceValida
  tionFlag>
<MQSISystemInteractionCheckFlag>False</MQSISystemInteractionCheckF
lag>
  <WorkFlowManagementFlag>True</WorkFlowManagementFlag>
    <WorkFlowQueueManager>FMCQM</WorkFlowQueueManager>
    <PublishFlag>True</PublishFlag>
    <OverrideFlag>True</OverrideFlag>
    <PublishErrorsFlag>True</PublishErrorsFlag>
    <PublishTopic>SyncCustomer</PublishTopic>
```

```
<WorkFlowDataStructureName>Test_ProcessTemplateExecute_1</WorkFlo
Data StructureName>
<WorkFlowProcessName>SetDestination</WorkFlowProcessName>
         <WorkFlowQueue>FMC.FMCGRP.EXE.XML</WorkFlowQueue>
   <WorkFlowSymbolic>Workflow1</WorkFlowSymbolic>
   <WorkFlowReplyToQueueManager>MQSIQM</WorkFlowReplyToQueueMan-
ager>
   <WorkFlowReplyToQueue>MQWF_END</WorkFlowReplyToQueue>
   <HubQueueManager>MQSIQM</HubQueueManager>
      <DefaultDestinationSymbolic>WorkFlowDefault</DefaultDestina-
tionSymbolic>
   <MessageTypeDependency></MessageTypeDependency>
   <TraceFlag>False</TraceFlag>
         <MQSIMessageEnabledFlag>True</MQSIMessageEnabledFlag>
      <WorkFlowParametersList>
         <Item>
            <ParameterName></ParameterName>
            <ParameterPath></ParameterPath>
            <DefaultValue></DefaultValue>
            <RequiredParameterFlag></RequiredParameterFlag>
       </Item>
</WorkFlowParametersList>
      <SystemInteractionList>
         <InteractionProblemsFlag>False</InteractionProblemsFlag>
         <Item>
            <SystemSymbolic>CIIS</SystemSymbolic>
           <RequiredInteractionFlag>True</RequiredInteractionFlag>
            <SystemMirror>CIISMirror1</SystemMirror>
            <SystemBackup>CIISBackup1</SystemBackup>
         </Item>
      </SystemInteractionList>
   </MessageProfile>
   <Session>
      <SessionId> </SessionId>
      <SessionState>Valid</SessionState>
</Session>
<SystemProfileArea>
      <SourceSystemProfile>
          <SystemSymbolic></SystemSymbolic>
```

```
        <StoreFlag></StoreFlag>
        <Language>10</Language>
    </SourceSystemProfile>
<DestinationSystemProfileList>
 <Item>
   <SystemSymbolic>CIIS1</SystemSymbolic>
   <MessageTypeName>AddParty</MessageTypeName>
   <StoreFlag>True</StoreFlag>
   <NextBackup>CIIS2</NextBackup>
 </Item>
      </DestinationSystemProfileList>
   </SystemProfileArea>
   <CRF_WorkArea>
       <MessageOffset>X</MessageOffset>
     </CRF_WorkArea>
  </MQSI_WorkArea>
```

Appendix F
MQSeries Workflow container structure

Description

This document describes the structures and content, and gives XML examples for, the container sections of the four WorkFlow messages that WMQI Enabler uses in communications between MQSI and WorkFlow. The container section is only part of a WorkFlow message. There is more content needed to create a valid WorkFlow message than this document discusses. Please refer to the WorkFlow application documentation for the structure of other areas of these messages.

Document changes

1. The `OriginalMessageId` field along with a description is now in the `CommonArea` template data structure.
2. The `ProcessReplyFlag` field along with a description is now in the OutgoingMessageArea template data structure.
3. The format for specifying multiple Systems in the `SystemInfoArea` was change from an array structure to numbered tags. The description was updated accordingly and examples are now given.
4. The format for specifying multiple Changes within the `FieldChanges` template was change from an array structure to numbered tags. The description was updated accordingly and examples are now given.
5. The format for specifying multiple Parameters within the `RequestedParameters` template was change from an array structure to numbered tags. The description was updated accordingly and examples are now given.
6. The `ProcessTemplateExecute`, `ActivityImplInvoke`, ActivityImplInvokeResponse, and ProcessTemplateExecuteResponse message examples have been updated to reflect changes (1 - 5) listed above. Also, the paths included in the Path tags have been update to reflect new API path rules.
7. A section titled WorkFlow Mapping Rules has been added to the document with a section on General rules.
8. The `Publish` field and its description have been added to the `CommonArea` section.

9. The NoDestinationFlag, MessageTemplateFlag, and NameMessage fields have been added to the OutgoingMessageArea along with descriptions for each.

Document conventions

Text in `monospace` font must be used (coded) in the exact form represented. All upper and lower case letters must be maintained. All occurrences and absences of space and other symbols must also be maintained.

Terminology

This section will list each term and its definition in the effort to reduce confusion. This document will use some terminology that is derived from an Object Oriented environment. While the terms will be defined as well as possible, please refer to literature on Object Oriented Terminology if you need a complete background.

container
This is the section of the WorkFlow message that is used to pass data between MQSI and WorkFlow.

data structure
This is an actual data structure use to define the WorkFlow container. Each data structure will need to be built in the WorkFlow build time GUI. See the WorkFlow application documentation for a more complete definition.

subdata structure
This is a data structure that is used to define parts (elements) of other data structures.

template data structure
This is a sub data structure that is never used alone to define the WorkFlow container. This is similar to the Object Oriented term "class". Also the term's short version (template) is used.

template data structure name
This refers to the name of a template data structure. Also the term's short version (template name) is used.

data structure instance
This refers to the element of a data structure that is defined by a template data structure. This is similar to the Object Oriented term "instance" or "class instance". Also the term's short version (instance) is used.

data structure instance name
This refers to the name of a data structure element that is defined by a template data structure. Also the term's short version (instance name) is used.

system
This refers to a front end or back end system (computer) that interacts with the hub.

system symbolic
This is a string that represents a system. The string is use to retrieve routing information from the SDR. WorkFlow places the system symbolic in the destinationLogicalId attribute of the Message tag to have a message routed to the system. Also the term's short version (symbolic) is used.

system active flag
This is a Boolean value that indicates whether or not a system is ready to receive messages. If the flag is True, then the system is currently active or up. If the flag is False, then the system is currently inactive or down.

A message should never be sent to an inactive system. The value of False is only placed in the system active flag of a system that is specified as NOT a required interaction in the message profile. This does not mean that in some uncommon circumstances that WorkFlow will never require interaction with these systems. It only means that it is possible to complete the process with out the interaction. If it is NOT possible for WorkFlow to complete a process with out interacting with a given system, then it should be listed in the message profile as required.

If MQSI receives a message that requires interaction with a system that is NOT currently active, then the message will be returned to the sender with an error or held until the system(s) required for interaction are active. This means that WorkFlow should never receive a message if a system that WorkFlow always uses is NOT active. Also the term's short version (active flag) is used.

Template data structures

CommonArea

This template data structure holds fields that are required for all messages. Only one version of this template is needed, because all of its fields are always used. The instance name given to an element defined by this template must always be CommonArea. The template built in the WorkFlow build time can have a different name.

This template data structure holds fields that are required for all messages. Only one version of this template is needed, because all of its fields are always used. The instance name given to an element defined by this template must always be CommonArea. The template built in the WorkFlow build time can have a different name.

`SessionId`	This field is a string that holds the id of the session that MQSI has assigned to a hub connection.
`ProcessId`	This field is a string that holds the id of the process that MQSI has assigned to the message type.
`OriginalMessageId`	This field is a string that holds the id of the original message that MQSI received from a front-end system, which is requesting a process to be started.
`MessageId`	This field is a string that holds the id of the message that MQSI has received. This field is only used by MQSI if the MessageName field of the `OutGoingMessageArea` is NOT provided.
`Publish`	This field is a string that holds the topic to which the message should be published. If the field is null, no publish action will be taken.

System

This is a template data structure that holds information pertaining to a single system that WorkFlow can communicate with to complete the executed process. Only one version of this template is needed, because all of its fields are always used. The instance name given to an element defined by this template must always be System. The template built in the WorkFlow build time can have a different name.

`Symbolic`	This field is a string that holds a system symbolic.
`ActiveFlag`	This field is a string that holds a system active flag.

SystemInfoArea

This template data structure holds information on the systems that workflow can use to complete the executed process. Multiple template data structures may have to be created to provide the variations in the content of this structure. The instance name given to an element defined by one of these templates must always be `SystemInfoArea` regardless of the name of the template

`SystemX` This is an array of elements of the template data structure called System, where X is a variable number of system(s) on which WorkFlow is expecting to receive information. WorkFlow must already know how each system is used by the order in which they appear or by their numbering (example: System1, System2, System3, …). The order and total number of systems are equivalent to that which appears in the message profile for the current message type.

DynamicParametersArea

This template data structure holds a variable number of parameters that the message profile specified for this message. The order and number of parameters is the same as appears in the message profile for the current message type. Each one of the parameters has the form described below. Multiple template data structures may have to be created to provide the variations in the content of this structure. The instance name given to an element defined by one of these templates must always be `DynamicParametersArea` regardless of the name of the template.

`X` Where X is the name of the field, which is a string that holds a parameter value. The name of the field and the value of the string are determined by two different methods. Each method is defined in the section describing the WorkFlow message in which it is used. The `DynamicParametersArea` template is used in the `ProcessTemplateExecute` and `ActivityImplInvokeResponse` messages.

Change

This template data structure holds information pertaining to one particular change to a field that WorkFlow is requesting MQSI to make to the outgoing message. Only one version of this template is needed, because all of its fields are always used. The instance name given to an element defined by this template must always be Change regardless of the name of the template. The template built in the WorkFlow build time can have a different name.

`Path` This field is a string that holds the path to a changing field.

`NewValue` This field is a string that holds the new value to place in the changing field.

FieldChanges

This template data structure holds a variable number of elements of the Change template. Each one of the parameters has the form described below. Multiple template data structures may have to be created to provide the variations in the content of this structure. The instance name given to an element defined by one of these templates must always be `FieldChanges` regardless of the name of the template.

`ChangeX` This is an array of elements of the template data structure called Change, where X is a variable number, which corresponds to a single field change. For example, if three changes were needed then you would see Change1, Change2, and Change3.

OutgoingMessageArea

This template data structure holds fields describing the requests WorkFlow is making of MQSI related to the outgoing message. Multiple template data structures may have to be created to provide the variations in the content of this structure. The instance name given to an element defined by one of these templates must always be `OutgoingMessageArea` regardless of the name of the template.

`MessageTemplateFlag` This element is a string that holds the text representation of a Boolean value. When its value is False, the message identified by the name within the element LoadMessage is a message named by WorkFlow previously. When its value is True, the message identified by the name within the element LoadMessage is a message template available for creating a new message. If this element is null then the value false is assumed.

`MessageName` This element is a string that holds the name of a message to be loaded and sent as the outgoing message. If this element is null then no loading is done and the last message received and identified by the MessageId element within the CommonArea will be sent as the outgoing message.

`ProcessReplyFlag` This element is a string that holds the text representation of a Boolean value. When its value is False no special processing is activated. When its value is True then logic is preformed by MQSI to correctly form and send the specified message as a reply to the system that sent the original message, which started the process.

`FieldChanges`	This element is an instance of the template data structure called FieldChanges. It specifies changes to make to the outgoing message.
`NoDestinationFlag`	This element is a string that holds the text representation of a Boolean value. When its value is False a message is sent out of the hub by the MQSI flows. When its value is True then the MQSI flows know to perform all requested processing of the specified message, but does not send it out of the hub.
`MessageTemplateFlag`	This element is a string that holds the name of the message template to load from the database. This message template is used for all processing.
`NameMessage`	This element is a string that holds the name to be given to the outgoing message after any specified changes have been made, so that the message can be referenced later.

Parameter

This template data structure holds fields describing a single parameter WorkFlow is requesting to be sent in the `ActivityImplInvokeResponse` message. Only one version of this template is needed, because all of its fields are always used. The instance name given to an element defined by this template must always be `Parameter` regardless of the name of the template. The template built in the WorkFlow build time can have a different name.

`Name`	This element is a string holding the name of one parameter that will appear in the DynamicParametersArea of the `ActivityImplInvokeResponse` message.
`Path`	This element is a string holding the path used to retrieve from the message the value of the parameter.

RequestedParameters

This template data structure holds information pertaining to the parameters WorkFlow is requesting to be sent in the `ActivityImplInvokeResponse` message. Multiple template data structures may have to be created to provide the variations in the content of this structure. The instance name given to an element defined by one of these templates must always be `RequestedParameters` regardless of the name of the template.

`ParameterX` This is an array of elements of the template data structure called Parameter, where X is a variable number, which corresponds to a single parameter that WorkFlow is requesting to be sent. For example, if three parameters were needed then you would see Parameter1, Parameter2, and Parameter3.

IncomingMessageArea

This template data structure holds fields describing the requests WorkFlow is making of MQSI related to the incoming message. Multiple template data structures may have to be created to provide the variations in the content of this structure. The instance name given to an element defined by one of these templates must always be `IncomingMessageArea` regardless of the name of the template.

`MessageName` This element is a string holding the name to be given to the incoming message so that is can be referenced later.

`WorkFlowDataStructureName`

This element is a string holding the name of the data structure that WorkFlow is expecting to see in the container section of the `ActivityImplInvokeResponse` message.

`RequestedParameters` This element is an instance of the template data structure called RequestedParameters.

ErrorInfoArea

This template data structure holds information pertaining to a single error MQSI or WorkFlow may have found while performing request or process respectively. Only one version of this template is needed, because all of its fields are always used. The instance name given to an element defined by this template must always be `ErrorInfoArea` regardless of the name of the template. The template built in the WorkFlow build time can have a different name.

`Error` This field is a string holding information describing the error that MQSI needs to send to WorkFlow in the ActivityImplInvokeResponse message or that WorkFlow needs to send to MQSI in the ProcessTemplateExecuteResponse message.

Messages

ProcessTemplateExecute

This message is sent from MQSI to WorkFlow to start a new WorkFlow process.

`CommonArea`	This element is an instance of the template data structure called CommonArea. Its three string elements are `SessionId`, `ProcessId`, and `MessageId`. For more details, see the CommonArea under the Template Data Structures section of this document.
`SystemInfoArea`	This element is an instance of the template data structure called SystemInfoArea. It holds a dynamic number of elements of the template data structure called System. For more details, see the SystemInfoArea under the Template Data Structures section of this document.
`DynamicParametersArea`	This element is an instance of the template data structure called DynamicParametersArea. It holds a variable number of elements. Each field's name and value is determined by referencing the message profile of the current message type. If the value of the field cannot be resolved by MQSI and the field is defined as not required in the message profile, then it is omitted. If a parameter is defined as required and the value cannot be resolved, then MQSI will declare an error and WorkFlow will never receive this message.

ActivityImplInvoke

This message is sent from WorkFlow to MQSI to perform a specified set of activities.

`CommonArea`	This element is an instance of the template data structure called CommonArea. Its three string elements are `SessionId`, `ProcessId`, and `MessageId`. For more details, see the CommonArea under the Template Data Structures section of this document.

`OutgoingMessageArea` This element is an instance of the template data structure called OutgoingMessageArea. Its two elements are `MessageName` and `FieldChanges`. For more details, see the OutgoingMessageArea under the Template Data Structures section of this document.

`IncomingMessageArea` This element is an instance of the template data structure called IncomingMessageArea. Its three elements are `MessageName`, `WorkFlowDataStructureName`, and `RequestedParameters`. For more details, see the IncomingMessageArea under the Template Data Structures section of this document.

ActivityImplInvokeResponse

This message is sent from MQSI to WorkFlow as a response to the ActivityImplInvoke message.

`CommonArea` This element is an instance of the template data structure called CommonArea. Its three string elements are `SessionId`, `ProcessId`, and `MessageId`. For more details, see the CommonArea under the Template Data Structures section of this document.

`SystemInfoArea` This element is an instance of the template data structure called SystemInfoArea. It holds a dynamic number of elements of the template data structure called System. For more details, see the SystemInfoArea under the Template Data Structures section of this document.

`DynamicParametersArea`

This element is an instance of the template data structure called `DynamicParametersArea`. It holds a variable number of parameters. Each parameter name and value is determined by referencing the data stored from the IncomingMessageArea of the `ActivityImplInvoke` message. If the value of the field cannot be resolved by MQSI, then MQSI will declare an error and WorkFlow will receive this message with an `ErrorInfoArea` describing the problem.

`ErrorInfoArea` This element is an instance of the template data structure called ErrorInfoArea. It can be omitted if an error has NOT occurred. The ErrorInfoArea template holds a string element called `Error`, which describes an

error that has occurred. For more details, see the ErrorInfoArea under the Template Data Structures section of this document.

ProcessTemplateExecuteResponse

This message is sent from WorkFlow to MQSI as a declaration of process completion.

`CommonArea`	This element is an instance of the template data structure called CommonArea. Its three string elements are `SessionId`, `ProcessId`, and `MessageId`. For more details, see the CommonArea under the Template Data Structures section of this document.
`ErrorInfoArea`	This element is an instance of the template data structure called ErrorInfoArea. It can be omitted if an error has NOT occurred. The ErrorInfoArea template holds a string element called `Error`, which describes an error that has occurred. For more details, see the ErrorInfoArea under the Template Data Structures section of this document.

Workflow mapping rules

General

1. The information in CommonArea must always be maintained to communicate with the MQSI flows. This means that it should always be mapped to the CommonArea of a data structure that is being sent to the MQSI flows. This holds for both the `ActivityImplInvoke` and `ProcessTemplateExecuteResponse` messages.
2. When a WorkFlow process terminates itself because of an error, the value(s) that were used to arrive at the decision should be sent to the MQSI flows. This is so that the flows can record the ending conditions of the process. To accomplish this, the value(s) should be mapped to the `Error` field of the `ErrorInfoArea` in the `ProcessTemplateExecuteResponse` message. This is the message that is sent to the MQSI flows via the process sink. Values that should be saved include the value of the `Error` field in an `ActivityImplInvokeResponse` message, if it caused a decision for termination.
3. When message naming is not used, careful mapping must be done to maintain the value of the `MessageId` of the `CommonArea`.

Examples

Workflow XML message examples

NOTE: *When an asterisk ("*") appears before and after a tag name it means the actual tag name is dynamic. When a period (".") appears at the same indention point the next line below a tag, it means that there can be more occurrences of that tag at the current level of nesting. Neither the asterisk nor the period should be included as part of an actual instance of this XML message. In the examples below all possible tags are included together in their respective messages even though they may never appear together in the same instance of a message. Again, only the container sections of the WorkFlow messages are included.*

ProcessTemplateExecute

```
        <*DataStructureNameHere*>
           <CommonArea>
              <SessionId>          </SessionId>
              <ProcessId>          </ProcessId>
              <OriginalMessageId>  </OriginalMessageId>
              <MessageId>          </MessageId>
              <Publish>           </Publish>
           </CommonArea>
           <SystemInfoArea>
              <System1>
                 <Symbolic>CIIS</Symbolic>
                 <ActiveFlag>True</ActiveFlag>
              </System1>
              <System2>
                 <Symbolic>CIIS2</Symbolic>
                 <ActiveFlag>False</ActiveFlag>
              </System2>
           </SystemInfoArea>
           <DynamicParametersArea>
              <*DynamicParameterName1*>
              </*DynamicParameterName1*>
              <*DynamicParameterName2*>
              </*DynamicParameterName2*>
              <*DynamicParameterName3*>
              </*DynamicParameterName3*>
              <*DynamicParameterNameN*>
              </*DynamicParameterNameN*>
           </DynamicParametersArea>
</*DataStructureNameHere*>
```

ActivityImplInvoke

```
        <*DataStructureNameHere*>
           <CommonArea>
              <SessionId>          </SessionId>
              <ProcessId>          </ProcessId>
              <OriginalMessageId>  </OriginalMessageId>
              <MessageId>          </MessageId>
              <Publish>           </Publish>
           </CommonArea>
           <OutgoingMessageArea>
           <MessageTemplateFlag>True</MessageTemplate-
           Flag>
           <MessageName>Activity12a</MessageName>
           <ProcessReplyFlag>False</ProcessReplyFlag>
           <FieldChanges>
                 <Change1>
                    <Path>(XML.tag)”Mes-
                    sage”.(XML.attr)”destinationLogi-
                    calId”</Path>
                    <NewValue>Party</NewValue>
                 </Change1>
                 <Change2>
                    <Path>
                       (XML.tag)”Message”.(XML.tag)”COM-
                    MAND”.(XML.tag)[1].
                       (XML.attr)”cmdStatus”
                    </Path>
                    <NewValue>notok</NewValue>
                 </Change2>
           </FieldChanges>
           <NoDestinationFlag></NoDestinationFlag>
           <NameMessage>Activity1Request</NameMessage>
           </OutgoingMessageArea>
           <IncomingMessageArea>
              <MessageName>Activity1Reply</MessageName>
              <WorkFlowDataStructure-
              Name>ActivityImplInvoke1</WorkFlowData-
              StructureName>
<RequestedParameters>
```

```
                  <Parameter1>
                     <Name>WFTransition</Name>
                     <Path>
                        XML.(XML.tag)”Mes-
                     sage”.(XML.tag)”COMMAND”.(XML.tag[1].
                        (XML.attr)”cmdStatus”
                     </Path>
                  </Parameter1>
                  <Parameter2>
                     <Name>PutTime</Name>
                     <Path>MQMD.PutTime</Path>
                  </Parameter2>
               </RequestedParameters>
            </IncomingMessageArea>
</*DataStructureNameHere*>
```

ActivityImplInvokeResponse

```
         <*DataStructureNameHere*>
            <CommonArea>
               <SessionId>          </SessionId>
               <ProcessId>          </ProcessId>
               <OriginalMessageId>  </OriginalMessageId>
               <MessageId>          </MessageId>
               <Publish>            </Publish>
            </CommonArea>
            <SystemInfoArea>
               <System1>
                  <Symbolic>CIIS</Symbolic>
                  <ActiveFlag>True</ActiveFlag>
               </System1>
               <System2>
                  <Symbolic>CIIS2</Symbolic>
                  <ActiveFlag>False</ActiveFlag>
               </System2>
            </SystemInfoArea>
            <DynamicParametersArea>
               <*DynamicParameterName1*>
               </*DynamicParameterName1*>
               <*DynamicParameterName2*>
               </*DynamicParameterName2*>
               <*DynamicParameterName3*>
               </*DynamicParameterName3*>
               <*DynamicParameterNameN*> </*DynamicParame-
               terNameN*>
            </DynamicParametersArea>
            <ErrorInfoArea>
               <Error>Bad path to parameter #1</Error>
            </ErrorInfoArea>
</*DataStructureNameHere*>
```

ProcessTemplateExecuteResponse

```
        <*DataStructureNameHere*>
           <CommonArea>
              <SessionId>          </SessionId>
              <ProcessId>          </ProcessId>
              <OriginalMessageId>  </OriginalMessageId>
              <MessageId>          </MessageId>
              <Publish>              </Publish>
           </CommonArea>
           <ErrorInfoArea>
              <Error>Unknown parameter value</Error>
           </ErrorInfoArea>
</*DataStructureNameHere*>
```

Appendix G
Notices

This information was developed for products and services offered in the U.S.A. and Europe. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS

FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories
Hursley Park
WINCHESTER, Hampshire
SO21 2JN
United Kingdom

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same

on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks or services of IBM Corporation in the United States or other countries or both:

IBM®

MQSeries®

DB2®

IAA®

Insurance Application Architecture®

BPM Workbench, BPM Monitor, and BPM Server are a trademarks of Holosofx, Inc. in the United States or other countries or both.

OAG is a trademark of the Open Architecture Group in the United States or other countries or both.

Other company, product, and service names may be trademarks or service marks of others.

Permission statement

Redistribution and use of this material for both commercial and noncommercial purposes are permitted subject to the below-stated conditions:

1. This Permission Statement shall be reproduced in its entirety in each copy of the material;
2. This material is provided AS IS without warranty of any kind, including but not limited to, any warranty of noninfringement or any warranty (express or implied) of merchantability or fitness for a particular purpose; and
3. The material may be modified provided
   a. Prior written notice of each modification is provided to the Interactive Financial eXchange Forum at the address listed below,

Interactive Financial Exchange Forum, Inc.
333 John Carlyle Street
Suite 600
Alexandria, VA 22314
U.S.A.

b. Any redistribution of modified materials shall be accompanied by a notice that modifications have been made and a clear description of the modifications, and

c. The party making the modifications assumes all responsibility for the consequences of the modifications.

Glossary

This glossary defines terms and abbreviations used in this book. If you do not find the term you are looking for, see the *Index* or the ***IBM Dictionary of Computing***, New York: McGraw-Hill, 1994.

A

Adapters
(1) A part that electrically or physically connects a device to a computer or to another device.

(2) A circuit board that adds function to a computer.

(3) Event Adapter: In a Tivoli environment, software that converts events into a format that the Tivoli Enterprise Console can use and forwards the events to the event server. Using the Tivoli Event Integration Facility, an organization can develop its own event adapters, tailored to its network environment and specific needs.

API: Application Programming Interface
(1) A software interface that enables applications to communicate with each other. An API is the set of programming language constructs or statements that can be coded in an application program to obtain the specific functions and services provided by an underlying operating system or service program.

(2) In VTAM, the language structure used in control blocks so that application programs can reference them and be identified to VTAM.

C

CRF: Cross Reference Function
This refers specifically to the storage system WMQI Enabler uses in order to keep track of creations of and attachments to UUID's.

D

DB2
An IBM relational database management system that is available as a licensed program on several operating systems. Programmers and users of DB2 can create, access, modify, and delete data in relational tables using a variety of interfaces.

DTD: Document Type Definition
The rules that specify the structure for a particular class of SGML or XML documents. The DTD defines the structure with elements, attributes, and notations, and it establishes constraints for how each element, attribute, and notation may be used within the particular class of documents. A DTD is analogous to a

database schema in that the DTD completely describes the structure for a particular markup language.

I

IAA: Insurance Application Architecture
IBM's business model for the insurance and financial services industry.

L

LDAP: Lightweight Directory Access Protocol
An open protocol that (a) uses TCP/IP to provide access to directories that support an X.500 model and (b) does not incur the resource requirements of the more complex X.500 Directory Access Protocol (DAP). Applications that use LDAP (known as directory-enabled applications) can use the directory as a common data store and for retrieving information about people or services, such as e-mail addresses, public keys, or service-specific configuration parameters. LDAP was originally specified in RFC 1777. LDAP version 3 is specified in RFC 2251, and the IETF continues work on additional standard functions. Some of the IETF-defined standard schemes for LDAP are found in RFC 2256.

M

MQSeries
Pertaining to a family of IBM licensed programs that provide message queuing services.

MQSI: MQSeries Integrator
It provides graphical tools for constructing how critical data or business events are handled, by visually connecting a sequence of processing function to dynamically manipulate and route messages, combine them with data from corporate databases, warehouse in-flight message data for auditing or subsequent analysis, and distribute information efficiently to business applications.

MQSWF: MQSeries Workflow
A business process management system, which facilitates the rapid development and management of the business processes that integrate the IT and organizational infrastructure of a company. It is a client/server system used to design, refine, document, and control a company's business processes using a graphical editor in one of its primary components to facilitate such modeling.

P

Party
Any person or organization that the insurance company has, or had, or may have a business interest in.

Property
A data value of a type.

Property tag
An XML tag representing a property of an IAA type (represented as an UML attribute).

Q

Queue
An 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. Queues of other types cannot contain messages: they point to other queues, or can be used as models for dynamic queues.

Queue Manager
A system program that provides queuing services to applications. It provides an application programming interface (the MQI) so that programs can access messages on the queues that the queue manager owns.

S

SQL: Structured Query Language
A programming language that is used to define and manipulate data in a relational database. It is often embedded in general purpose programming languages.

T

Tag
An XML construct <Tag....>.

Type tag
An XML tag representing an IAA type.

U

UUID: Universally Unique Identifier
This is a key used by the WMQI Enabler to uniquely identify the entities which outside systems need to reference.

X

XML
eXtensible Markup Language. XML is a markup language for message definition, and is an open and public domain standard. XML is a subset of SGML designed for easy implementation in commercial and web environments.

XML attribute (or just attribute)
Appears in an opening tag, used to specify values in the tag. <Tag attribute='val'...>

Index