IBM MQSeries® Adapter for Secure Financial Messaging Gateway

Installation and Programming Guide

Version 1 Release 1

SH12-6731-00

IBM MQSeries® Adapter for Secure Financial Messaging Gateway

Installation and Programming Guide

Version 1 Release 1

SH12-6731-00

Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix D. Notices" on page 119.

First Edition, May 2001

This edition applies to Version 1 Release 1 of IBM MQSeries Adapter for Secure Financial Messaging Gateway (5799-GKZ) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2001. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book v

Chapter 1. Introducing MQSeries Adapter 1

Process initiation and message flow 3
- Message flow when using InterAct 3
- Message flow when using local file transfer . . . 4
- Message flow when using FileAct 6
- Message flow when using the client hub 6
- Message flow when using the server request handler 7

Synchronous and asynchronous calls 8
Portability 9
Client hub and server request handler 9
- Starting a client hub or server request handler . . 9
- Stopping a client hub or server request handler 10

Data conversion 11
Authorization checking 13
- Levels of authorization checking 14
- Example of using authorization checking . . . 15
- Determining which RACF profile controls access to a protected resource 15

Chapter 2. Planning for and installing MQSeries Adapter 17

Planning for MQSeries Adapter. 17
- Hardware requirements 17
- Software requirements. 17

Installing MQSeries Adapter. 17
- Connect MQSeries Adapter to SAG 18
- Copy DTD files to MVS 18
- CICS and IMS dependencies and restrictions . . 18
- Implementing authorization checking. 20

Chapter 3. Creating client and server profiles 21

Chapter 4. Writing MQSeries Adapter clients and servers, and SwCallback functions. 25

Data structures 25
- AGHResponse 25
- AGHLFTCmdParm 26
- AGHFileHeader 27

| XML messages 28
Sample programs 29

Chapter 5. MQSeries Adapter C++ classes and methods 31

AGHClient 31
- Constructors 31
- Methods 32

AGHServer 39
- Constructors 40
- Methods 40

Chapter 6. MQSeries Adapter functions for C and COBOL 49

AGHClientGetConditionCode—Get condition code for client 50
- Format for C 50
- Coding examples 50

AGHClientGetErrorMessage—Get error message for client 51
- Format for C 51
- Coding examples 51

AGHClientSetClientName—Set a client name . . . 52
- Format for C 52
- Coding examples 52

AGHClientSetUserId—Set a client user ID 54
- Format for C 54
- Coding examples 54

AGHFCall—Initiate a FileAct transfer 56
- Format for C 56
- Coding examples 56

AGHLFTCmd—Issue an LFT command 58
- Format for C 58
- Coding examples 58

AGHServerClientHub—Start a client hub 61
- Format for C 61
- Coding examples 61

AGHServerGetConditionCode—Get condition code for server 62
- Format for C 62
- Coding examples 62

AGHServerGetErrorMessage—Get error message for server 63
- Format for C 63
- Coding examples 63

AGHServerInit—Initialize a server. 64
- Format for C 64
- Coding examples 64

AGHServerRelease—Release the server's message buffer 65
- Format for C 65
- Coding examples 65

AGHServerReply—Place a response in the reply-to queue 67
- Format for C 67
- Coding examples 67

AGHServerRequestHandler—Start a server request handler. 69
- Format for C 69
- Coding examples 69

AGHServerRetrieve—Retrieve a request 70
- Format for C 70
- Coding examples 70

AGHServerSetUserId—Set a user ID for a server . . 72

© Copyright IBM Corp. 2001

Format for C 72
Coding examples 72
AGHServerTerm—Terminate server 74
Format for C 74
Coding examples 74
SwACall—Initiate an asynchronous InterAct transfer 75
Format for C 75
Coding examples 75
SwAWait—Retrieve a response from an asynchronous call 77
Format for C 77
Coding examples 77
SwCall—Initiate a synchronous InterAct transfer . . 79
Format for C 79
Coding examples 79
SwXmlBufferFree—Free client buffer 81
Format for C 81
Coding examples 81

Chapter 7. User-written functions . . . 83
SwCallback 83
Coding examples 83
AppXmlBufferFree—Free server buffer 85
Format 85
Variables 85
Coding examples 85

Chapter 8. Message logging and tracing 87
Message logging. 87
Administering data in the system logger stream 88
Tracing 90

Appendix A. Messages and codes . . . 95
Messages 95
Condition codes 104
SWIFT status codes 105
Return codes 105

Appendix B. Process flows 107

Appendix C. Considerations when implementing authorization checking . 113
Authorization checking SVC routine. 113
Installation-defined RACF classes. 114
RACF class definition source 114
Generating the ICHRRCDE. 115
Activating installation-defined classes 116
Defining AGH.RS 117
Permit a user to access an application resource 117

Appendix D. Notices 119
Trademarks 120

Glossary of terms and abbreviations 121

Bibliography. 125

Index 127

About this book

This manual describes how to install, customize, and write programs that use IBM MQSeries Adapter for Secure Financial Messaging Gateway, abbreviated to MQSeries Adapter in this manual.

Readers of this manual should be familiar with the information contained in the following manuals:

- *SWIFTAlliance Gateway Developer Guide Release 1.2.0*
- *SWIFTAlliance Gateway Interface Specification Release 1.2.0*
- *SWIFTAlliance Gateway MQHA Application Programming Guide Release 1.2.0*
- *SWIFTAlliance Gateway MQHA Installation and Configuration Release 1.2.0*
- *SWIFTNet Link User's Guide*

© Copyright IBM Corp. 2001

Chapter 1. Introducing MQSeries Adapter

IBM MQSeries Adapter for Secure Financial Messaging Gateway, abbreviated to MQSeries Adapter in this manual, provides an application programming interface (API) that lets programs running on OS/390® systems exchange messages via the SWIFT Secure Internet Protocol Network (SIPN). Application programmers who write programs that request or provide services via the SIPN do not want to have to deal with communication-layer processes. The MQSeries Adapter API shields them from such processes.

A requesting application program (the *client*) located on an OS/390 system uses functions provided by MQSeries Adapter to pass requests, via MQSeries queues, to an instance of SWIFTAlliance Gateway (SAG). SAG, in turn, uses SWIFTNet Link (SNL) to route each request to its destination via the SIPN.

Inbound messages are received from the SIPN by an SAG and are forwarded to the receiving OS/390 system, where they are put into an MQSeries request queue. This triggers a responding application program (the *server*), which uses MQSeries Adapter to retrieve the request from the request queue, processes the request, and uses MQSeries Adapter to send its response back to the client. Because the MQSeries queues are located on the OS/390 system, only one MQSeries queue manager (which runs on the OS/390 system) is needed.

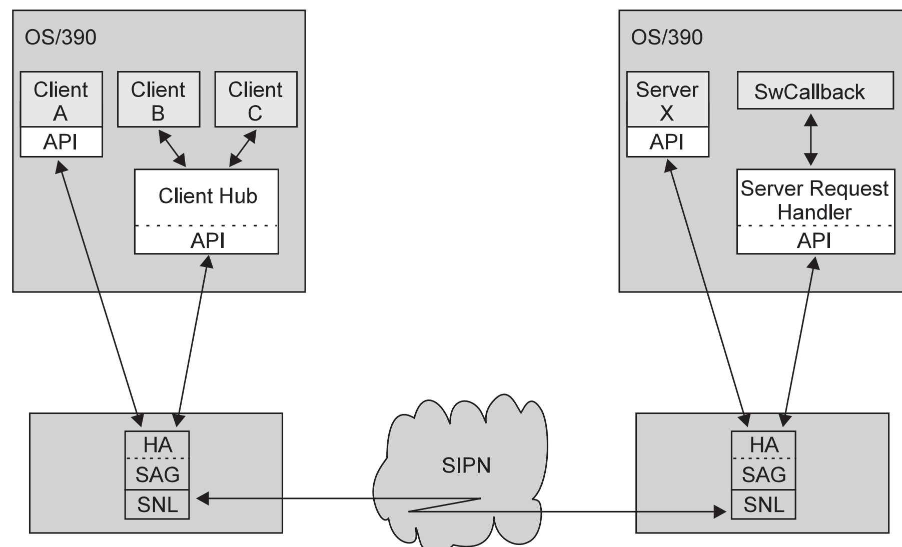

Figure 1. The MQSeries Adapter API lets application programs exchange messages via the SIPN

In addition to the dynamic link libraries (DLLs) that comprise the API, MQSeries Adapter provides the following programs to help you simplify your request processing:

MQSeries hub for client requests
: Instead of using the MQSeries Adapter API directly, clients can instead place their requests into an MQSeries queue that serves as their site's single point of access to the SIPN. MQSeries Adapter provides a program called the MQSeries hub for client requests (abbreviated to *client hub*) that

can then act as a single, centralized message processor. It handles the sending of requests and the routing of the responses back to the clients, while taking advantage of MQSeries Adapter's authorization checking and logging capabilities. Processing messages centrally in this way reduces the complexity of each individual client. It also increases flexibility, because the MQSeries queues that are used can be accessed not only from the system on which they are located, but from other systems as well.

Server request handler
If you prefer, instead of writing your own server, you can run a program provided by MQSeries Adapter called a *server request handler*. A server request handler continually retrieves requests from the request queue at its site and, for each request, calls a user-written function called *SwCallback* that processes each request according to your needs.

MQSeries Adapter supports:

Different environments
Programs that use the MQSeries Adapter API can run as MVS™ batch jobs, or in CICS® or IMS™ environments.

Basic and SNL messages
MQSeries Adapter can handle both basic messages (character strings) and SWIFTNet Link (SNL) messages (XML strings that conform to the rules established by S.W.I.F.T.).

InterAct and FileAct services
Programs that use MQSeries Adapter can take advantage of both SWIFT InterAct services (to send messages) and SWIFT FileAct services (to transfer files).

Local file transfer
Programs can use functions provided by MQSeries Adapter to put files onto an SAG workstation (for example, in preparation for a FileAct transfer), or to get, list, or delete such files.

MQSeries Adapter provides:

Authorization checking
MQSeries Adapter can be set up to use an external security manager (ESM) such as Resource Access Control Facility (RACF®) to control whether a client or server is granted access to the information needed to send messages to certain destinations, or to act on behalf of certain persons or institutions.

Message logging
To satisfy your auditing requirements, whenever MQSeries Adapter sends or receives a message, it can write a copy of the message, plus a header containing logging information, to a system logger stream, sequential data set, or both.

Data conversion
MQSeries Adapter converts XML messages as needed. It also converts messages to and from:

- Client and server applications that use EBCDIC code pages such as ibm-1047 or ibm037-s390
- SAG, which uses UTF-8 encoding

Additionally, clients and servers can use the built-in conversion function provided by MQSeries to translate basic messages from ASCII to EBCDIC, and vice versa.

Process initiation and message flow

When you start a client or server, for example by running a batch job or by executing a transaction in a CICS or IMS environment, one of the first things MQSeries Adapter does is read the profile named in the job and extract the parameters for the specified client or server. These parameters determine such things as:

- The name of the correspondent (message partner)
- Whether to convert the message data
- The names of the MQSeries queue manager and queues
- Whether message logging is to be activated

For a complete list of these parameters and their descriptions, see “Chapter 3. Creating client and server profiles” on page 21.

The subsequent message flow between clients and servers depends on the nature of the service that is to be performed.

Message flow when using InterAct

Figure 2 shows a typical message flow between clients and servers using InterAct service with MQSeries Adapter.

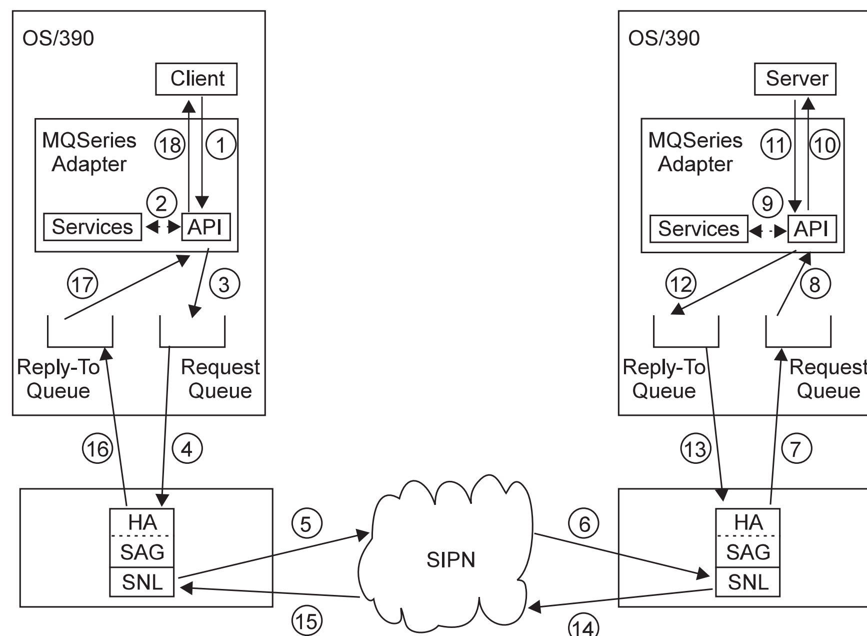

Figure 2. Message flow using InterAct when the client uses a local request queue

The client uses the functions provided by the MQSeries Adapter API (1) to place a request into the request queue at its site (3), while taking advantage of the message logging and authorization-checking services provided by MQSeries Adapter (2). For example, MQSeries Adapter can check whether the user ID associated with the client is authorized to send messages to the correspondent. An SAG at the client’s

site uses its MQSeries Host Adapter (HA) component to retrieve the request (4) and passes it to SNL, which routes it via the SIPN (5) to an SNL and SAG at the server's site (6). The SAG at the server's site places the request into the request queue at that site (7). The server uses the functions provided by the MQSeries Adapter API to retrieve the request from the request queue (8), while taking advantage of the message logging services provided by MQSeries Adapter (9). After receiving (10) and processing the request, the server uses the MQSeries Adapter API (11) to place its response into the reply-to queue at its site (12), taking advantage of the message logging and authorization checking services provided by MQSeries Adapter (9). An SAG at the server's site retrieves the response (13) and routes it via the SIPN (14) to an SAG at the client's site (15). The SAG places the response into the client site's reply-to queue (16). The client uses the MQSeries Adapter API (17) to retrieve the response from the reply-to queue (18), again taking advantage of the services provided by MQSeries Adapter (2).

The client need not run on the OS/390 system where the request queue is located. Instead, request and response messages can be passed between MQSeries message queue managers (MQMs) running on different systems, as shown in Figure 3.

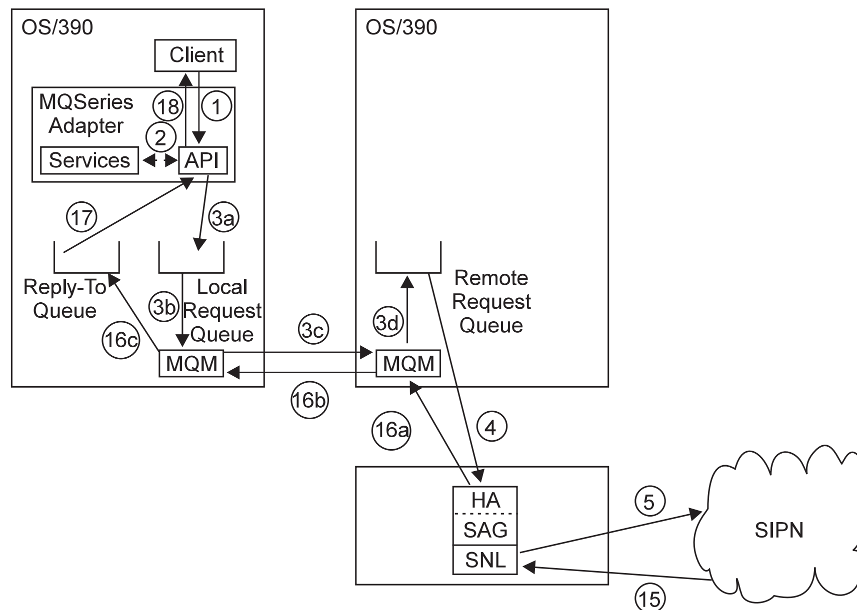

Figure 3. Message flow using InterAct when the client uses a remote request queue

The client uses the functions provided by the MQSeries Adapter API (1) to place a request into the request queue at its site (3a). This queue is a local definition of a remote queue. The local MQM retrieves the request from this queue (3b) and passes it to the remote MQM (3c), which places it into a remote request queue (3d). An SAG at the client's site retrieves the request (4), and processing continues as described in steps 5 through 15 in Figure 2. After the response arrives at the client's site, the SAG at that site passes it to the remote MQM (16a), which passes it to the local MQM (16b), which places it into the reply-to queue (16c). The client uses the MQSeries Adapter API (17) to retrieve the response from the reply-to queue (18).

Message flow when using local file transfer

MQSeries Adapter provides local file transfer (LFT) functions that you can use to transfer files from an OS/390 to the workstation on which SAG runs, or vice-versa.

Usually this is done in preparation for transferring the file to a remote system via the SIPN.

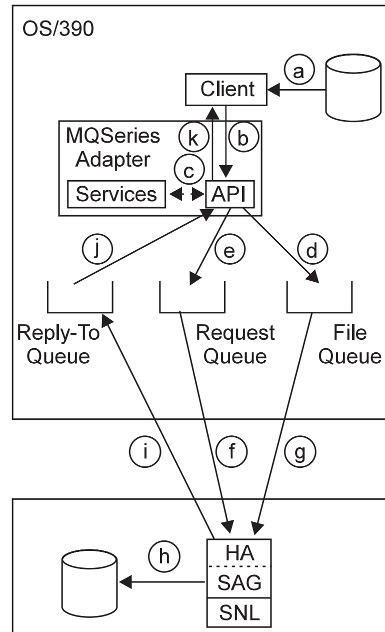

Figure 4. Message flow for a local file transfer

The client retrieves file data from a file (a), then uses the LFT PUT command provided by the MQSeries Adapter API (b) to place file data into its system's file queue (d), while taking advantage of the logging and authorization-checking services provided by MQSeries Adapter to log and check the LFT request (c). MQSeries Adapter then places a request containing the PUT command into the request queue (e). An SAG instance at the client's site retrieves the request (f), retrieves the corresponding file data (g), writes the data into the local file named in the PUT command (h), and writes a response to the reply-to queue (i). The client uses the MQSeries Adapter API to retrieve the response (j,k). To transfer the file to its destination, the client must then issue a FileAct request.

In addition to transferring files, you can also use MQSeries Adapter to:

- List all the files in the file directory on the SAG system
- Delete a file in the file directory on the SAG system
- Get a file from the file directory on the SAG system

For more information about the file directory, refer to *SWIFTAlliance Gateway MQHA Installation and Configuration Release 1.2.0*.

Message flow when using FileAct

The message flow when using FileAct service with MQSeries Adapter is similar to that using InterAct, except that after the SAG at the client's site receives the response (step 15 in Figure 2 on page 3), it not only places the response into the reply-to queue (step 16 in Figure 2), it also issues a *begin file transfer* message to the server request queues for file transfer events at both sites (a1 and a2,a3,a4), and begins the file transfer (b1,b2,b3,b4). When the file transfer is complete, the SAG at the server's site issues an *end file transfer* message to the server request queues at both sites (c1 and c2,c3,c4); this marks the end of the file transfer.

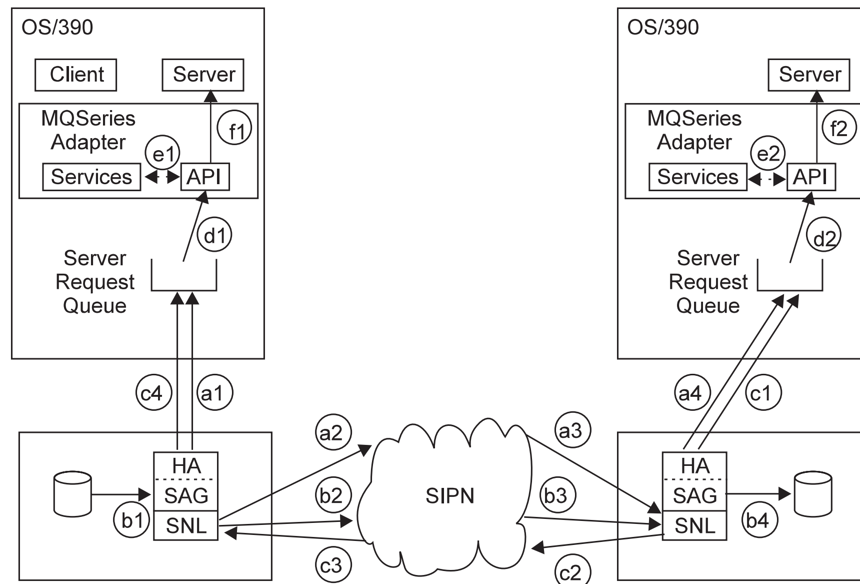

Figure 5. Message flow using FileAct

To access the information in the server request queues (for example, if the server needs to know when the file transfer is complete so that it can process the transferred data), a server program at the client or server site can use the *retrieve* server function to retrieve the *end file transfer* from the server request queue (d1,f1 and d2,f2). As always, the server can take advantage of the message logging service provided by MQSeries Adapter (e1 and e2).

Message flow when using the client hub

Rather than having clients use MQSeries Adapter functions, you can instead arrange for them to place their requests in a central MQSeries request queue, and use an MQSeries Adapter program called a client hub to handle the sending of requests and the routing of the responses back to the clients. The message flow when using the client hub is similar to that using InterAct or local file transfer (LFT), except that the clients use a central request queue and their own reply-to queues instead of using MQSeries Adapter directly.

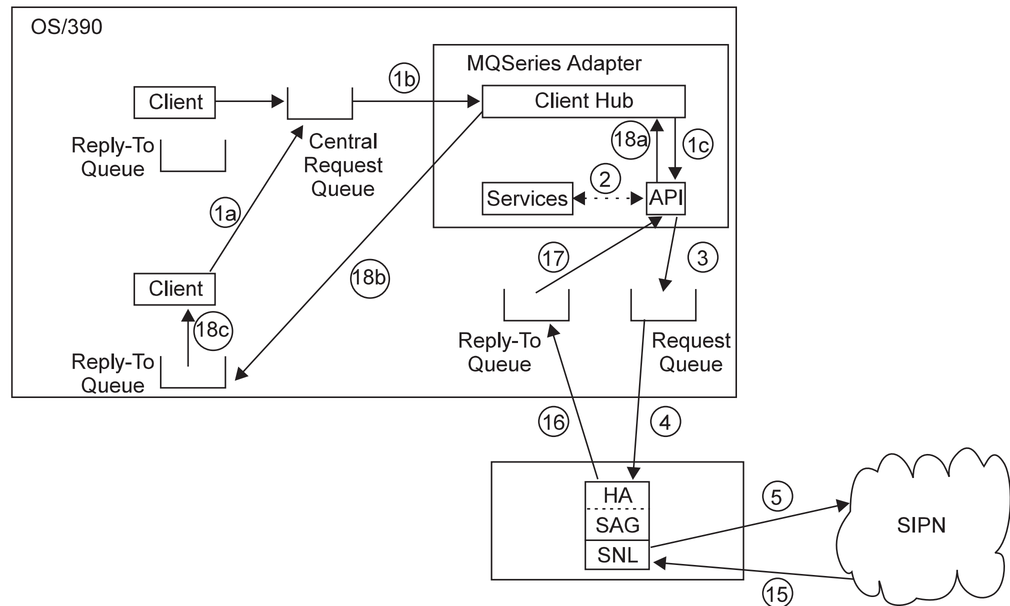

Figure 6. Message flow using the client hub

For an InterAct request, a client places the request into the central request queue (1a). The client hub retrieves requests from the central request queue (1b) and uses the functions provided by the MQSeries Adapter API (1c) to place a request into the request queue at its site (3). Processing continues as described in steps 4 through 17 in Figure 2 on page 3. After a response arrives at the client's site, the client hub uses the MQSeries Adapter API to retrieve it (18a) and place it into the reply-to queue of the corresponding client (18b), after which the client can retrieve it (18c).

The client hub can also be used for LFT commands. This way, only one program (the client hub), not each of your back-office client applications, needs to use MQSeries Adapter. LFT commands are indicated by the MQSeries message type MQMT_APPL_FIRST (65536). A separate queue (called a *file queue*) is used to transfer the files. The correlation between an LFT command and a file in the file queue is established by copying the MQSeries message ID of the file to the MQSeries correlation ID of the request containing the LFT command.

Message flow when using the server request handler

At the server's site, you can run a program provided by MQSeries Adapter called the *server request handler*, which continually retrieves requests from the request queue and, for each request, calls the SwCallback function. SwCallback is a user-written function that must be located in the DLL AGHCASCB, and that processes requests according to your needs.

The message flow when using the MQSeries Adapter server request handler is similar to that using InterAct, except that instead of a user-written server retrieving and processing the request, the server request handler retrieves it and calls a function to process it.

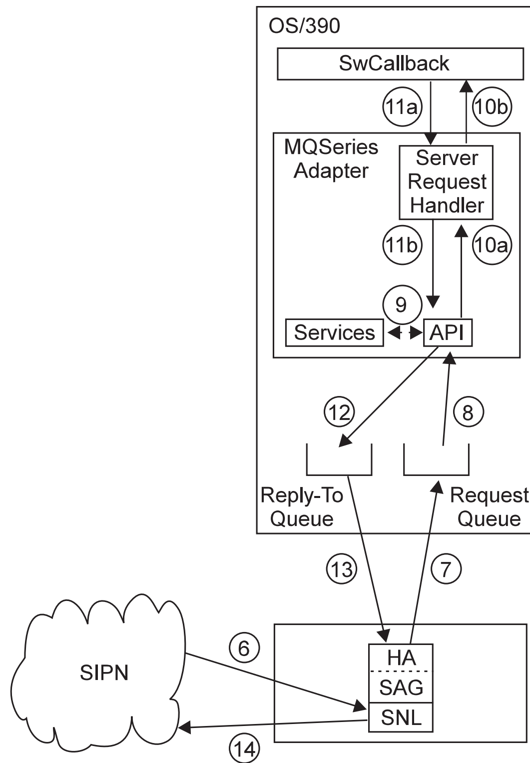

Figure 7. Message flow using the server request handler

Processing proceeds as described in steps 1 through 7 in Figure 2 on page 3. After the request arrives in the request queue at the server request handler's site, the server request handler uses the MQSeries Adapter API to retrieve the request (8) and pass it to the server request handler (10a). It then calls the user-written SwCallback function (10b), which processes the request, and passes its response back to the server request handler (11a). The server request handler uses the MQSeries Adapter API (11b) to place the response in the reply-to queue (12), and processing continues as described in steps 13 through 18 in Figure 2.

Synchronous and asynchronous calls

When sending an InterAct request, a client can issue either a synchronous or an asynchronous call:

Synchronous call (SwCall)
Control is given back to the client after MQSeries Adapter retrieves the response and passes it to the client (step 17 in Figure 2 on page 3). This process flow is shown in Figure 25 on page 107.

Asynchronous call (SwACall)
Control is given back to the client after the request has been put into the MQSeries request queue (step 3 in Figure 2 on page 3), and a message handle is passed back to the application. To retrieve the corresponding response, the program issues an SwAWait call specifying this message handle. After issuing an SwAWait call, control passes to MQSeries Adapter; control is given back to the client after either the response has been retrieved or a time-out has occurred. This process flow is shown in Figure 26 on page 108.

Portability

The MQSeries Adapter API is compatible with the API provided by SAG. As a result, application programs that currently run on a workstation and access SAG directly can be ported to an OS/390 system and can instead access SAG via MQSeries Adapter. Depending on how you choose to run your program (as a batch job, or as a CICS or IMS transaction), minor code changes might be required. The sample programs listed in "Sample programs" on page 29 illustrate such changes.

Client hub and server request handler

MQSeries Adapter provides two ready-to-use programs:

Client hub
As an alternative to writing your own client, you can use the client hub provided by MQSeries Adapter. For InterAct requests, all your application program needs to do is place standard SWIFT requests in an MQSeries queue and receive the responses in its reply-to queue. Using LFT commands with a client hub is somewhat more complicated, because the client must perform additional processing in order to provide the client hub with the data it needs to correlate a request with the corresponding file data (this is illustrated by the sample AGHCAAF2).

A sample client hub is defined in the profile SERVER02. A job to start it is contained in the member AGHMZRS1, which executes the program AGHCAAS1.

Server request handler
A server request handler is a program that retrieves requests from the request queue, passes them to the SwCallback function for processing, and places the responses generated by SwCallback into the reply-to queue.

Both a client hub and a server request handler will run as long as there are messages in its input queue. If its queue becomes empty and no messages arrive within the time-out interval specified in its profile, it checks in its profile whether the console interface is to be used:

- If so (Console=Y), it continues to run until the operator terminates it
- If not (Console=N), it stops

Starting a client hub or server request handler

Both the client hub (see Figure 6 on page 7) and the server request handler (see Figure 7 on page 8) are started using the same program. Whether the program starts a client hub or a server request handler is determined by a parameter that you specify when you run the program. Two forms of this program are located in the MQSeries Adapter sample library SAGHSAMC:

- AGHCAAS1 (written in C++)
- AGHCAAS2 (written in C)

This program accepts two parameters, which are specified following the slash (/) in the parameter string of the EXEC statement:

- The first parameter specifies the name of the client hub or server request handler. This name must correspond to a label in the profile data set.
- The second parameter can be one of the following:

REQUESTHANDLER
Starts a server request handler.

CLIENTHUB
Starts a client hub.

LOOP Starts a loop server, which is a simple server that merely returns received data. This is provided for test purposes only, for example to verify that the installation was successful.

The following is an example of an EXEC statement of a batch job used to start a client hub that uses the profile with the name SERVER02 (refer also to the sample job AGHMZRS1, which is located in SAGHRUNS).

```
//MRC$CLT  JOB ...
//GOSTEP   EXEC PGM=AGHCAAS1,REGION=0M,
//         PARM='POSIX(ON),ENVAR(''TZ=CET-1'')/SERVER02 CLIENTHUB'
```

Figure 8. EXEC statement of a batch job used to start a client hub

The POSIX(ON) run-time option and timezone need to be set only if you use the console interface (see “Stopping a client hub or server request handler”).

When running this program as a:

- Server request handler, there is no need to use the MQSeries Adapter API calls in your server; it is sufficient to implement the SwCallback function in a DLL that is called by the server request handler. This DLL must have the name AGHCASCB, and must implement the SwCallback function according to the specifications defined by S.W.I.F.T.
- Client hub, no additional coding effort is needed; only the profile parameters for the client hub must be set. The client hub profile definitions are identical to those for a server, except that the client hub uses an additional parameter (ClientName) to specify the name of a client profile in the same data set. This client profile contains the parameters that are used by the client.

Stopping a client hub or server request handler

You can use the system console to terminate a running client hub or server request handler, provided all the following are true:

- The parameter Console=Y was set for that client hub or server request handler in its profile.
- The process user has an OMVS segment.
- The batch job used to start the client hub or server request handler sets the POSIX(ON) run-time option (see Figure 8).

Make sure that the timezone information is also set correctly (an example is shown in Figure 8), otherwise the timestamp in the log might be incorrect.

If you specify in the client hub or server request handler’s profile:

Console=N The client hub or server request handler terminates as soon as its request queue has been empty for the interval specified for its TimeOut parameter (this parameter is described in

Table 1 on page 22). The LOOP function always terminates as soon as the request queue is empty, regardless of the setting of the Console profile parameter.

Console=Y
The client hub or server request handler remains active until it is stopped. To stop a client hub or server request handler, issue the MVS MODIFY command from the operator console. The value of this command's APPL parameter determines how the program is to terminate:

APPL=STOP
The program terminates immediately after the current request message has been processed.

APPL=SHUT
The program terminates as soon as its request queue has been empty for the interval specified for the TimeOut parameter in its profile (this parameter is described in Table 1 on page 22).

For example, to use the command in the TSO SDSF application to immediately stop a client hub with the name MRC$CLT, enter the MODIFY command as follows:

```
COMMAND INPUT ===> /f MRC$CLT,APPL=STOP

NP   JOBNAME  STEPNAME PROCSTEP JOBID    OWNER    C POS DP REAL PAGING
     MRC$CLT  GO                JOB03735 MRC      A IN  C3 2362   0.00
     .
     .
     .
```

The program responds with a message that indicates when it expects to terminate:

```
COMMAND INPUT ===>
RESPONSE=SYS1
BPXM023I (MRC)
AGH1132I: Request to terminate server request handler or client hub accepted,
          remaining wait time in seconds: 21
```

If the console interface is not active (that is, if Console=N was set in the request handler's profile), the system messages will indicate this:

```
RESPONSE=SYS1      IEE341I MRC$CLT          NOT ACTIVE
RESPONSE=SYS1      IEE342I MODIFY   REJECTED-TASK BUSY
```

In this case, you must terminate the job by some other means, for example an operator-initiated cancel command.

Data conversion

When necessary, MQSeries Adapter converts (or arranges for MQSeries to convert) the data that comprises the message being transferred. MQSeries Adapter differentiates among the following types of messages:

- Basic messages (simple strings encoded in EBCDIC format)
- SNL messages encoded in EBCDIC format, for example containing the EBCDIC string `<?xml version="1.0" encoding="IBM-1047"?>`

- SNL messages encoded in UTF-8 format (the S.W.I.F.T. standard), for example containing the ASCII string `<?xml version="1.0" encoding="UTF-8"?>`

When discussing data conversion, the following situations must be considered separately:

Clients sending requests, servers sending responses, or clients receiving responses
The MQSeries Adapter determines what type of data conversion to perform based on the contents of the message. If the message contains the beginning of an XML encoding declaration (that is, the string **`<?xml`**) in either EBCDIC or UTF-8, MQSeries Adapter treats the message as an SNL message. In all other cases, it treats the message as a basic message (that is, a simple EBCDIC string). For each request a client sends, it assumes the response will have the same format. For each request a server receives, it creates a response with the same format.

Servers receiving requests
Because the contents of the request message are not known to MQSeries Adapter at the time when it must decide which data conversion method to use (at that time the request is still in the request queue), the server must specify the data conversion method in advance. It does this via the **Convert** profile parameter, which is described in Table 1 on page 22. The value of this parameter must correspond to the SAG settings that determine whether basic or SNL messages are to be transferred.

The conversion processing for the three possible scenarios is shown in Figure 9, Figure 10, and Figure 11. Note that because each message format requires a special type of data conversion, and because a server can perform only one type of data conversion, there must be a separate server for each message format used. In this case, each server has its own request queue, and SAG must be configured so that it places a message of a particular format into the appropriate request queue.

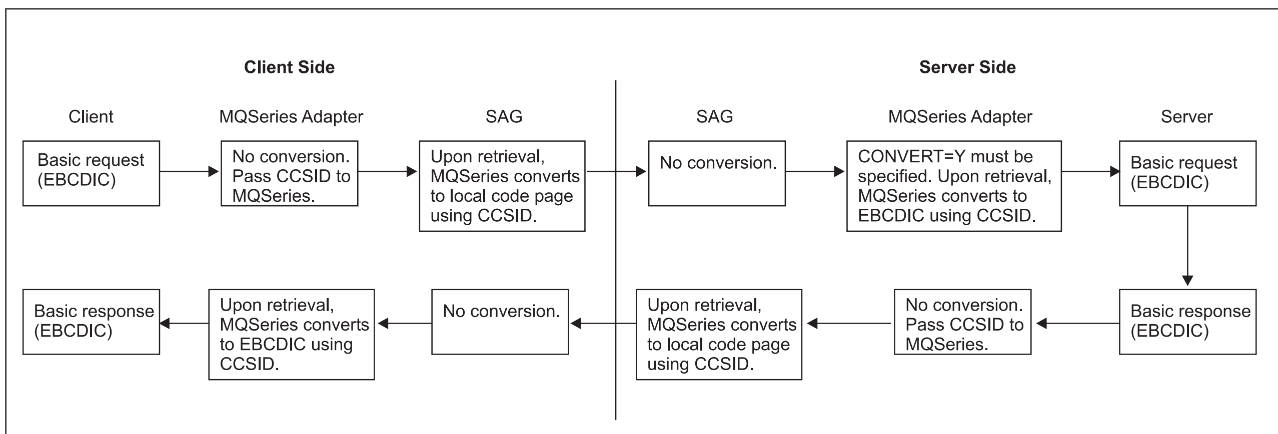

Figure 9. Basic messages (simple strings)

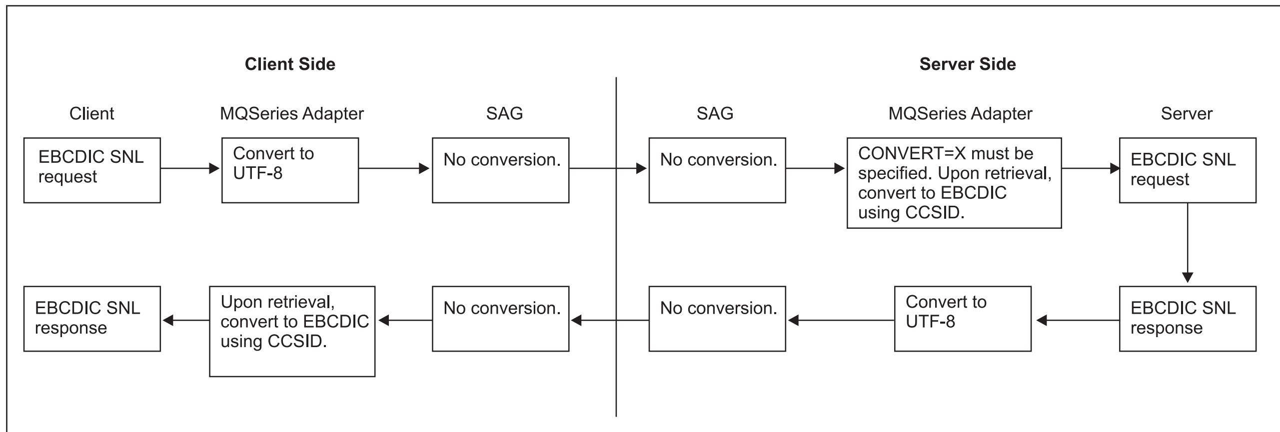

Figure 10. SNL messages encoded in EBCDIC format

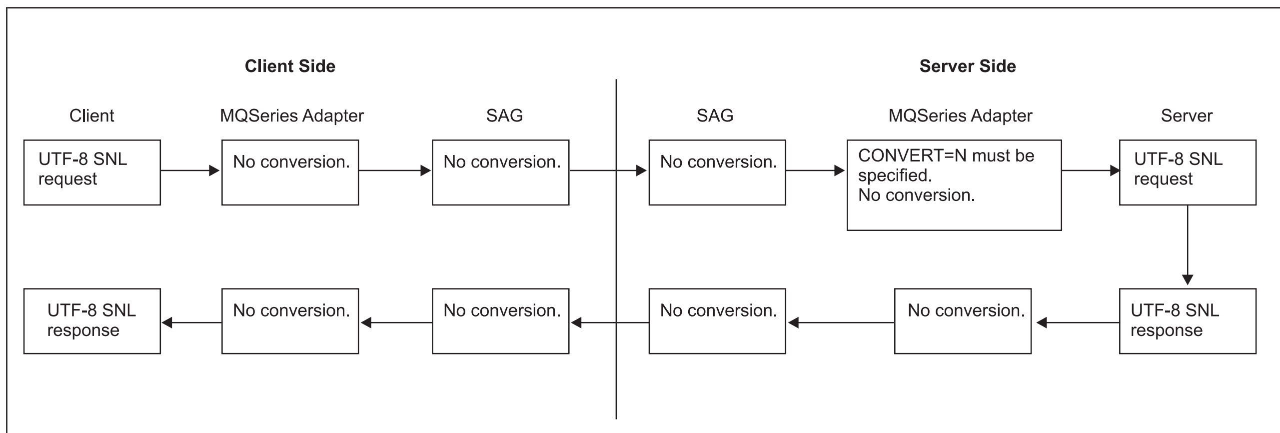

Figure 11. SNL messages encoded in UTF-8 format

For basic messages and EBCDIC SNL messages, the CCSID parameter specified in the client or server profile determines which EBCDIC code page is to be used for conversion (see Table 1 on page 22). For basic messages MQSeries carries out the conversion, and MQSeries Adapter passes the name of the code page to be used to MQSeries via the CodedCharacterSetId field of the MQMD. Note that MQSeries conversion does not support all code page combinations. If the code page combination you need is not supported, consider using the CCSID parameter to specify a different code page, but one that contains the characters used in your messages (for example, the MQSeries code page 500).

Authorization checking

The SAG table with the name **correspondents.dat** contains a series of entries, each of which has a unique *correspondent name*. Each correspondent name is associated with a list of *distinguished names* (DNs). The DNs used by MQSeries Adapter are:

Requestor
The name of the institution issuing the request

Responder
The name of the institution to which the request is to be sent, and from which the response is to come

Sign
The name of the person or institution on whose behalf the request is issued

A user who is authorized to use a correspondent name is automatically authorized to use the corresponding DNs. For SNL messages, and when running as a batch job or in an IMS environment, you can also authorize users for individual DNs of the types listed above. MQSeries Adapter uses RACF resource profiles in proprietary RACF resource classes to control a client user's access to correspondent names and DNs.

The correspondent name is specified in the client's MQSeries Adapter profile; the DNs are extracted from SNL messages. There must be a resource class defined in RACF for correspondent names, and for requestor, responder, and sign DNs. The resource class names used by MQSeries Adapter all start with **AGH$**:

AGH$CORN For correspondent names

AGH$RQDN For requestor DNs

AGH$RSDN For responder DNs

AGH$SGDN For sign DNs

Levels of authorization checking

The MQSeries Adapter authorization checking service is itself controlled by RACF definitions. These definitions are specified in the profile AGH.RS in the FACILITY class. If AGH.RS is not defined, MQSeries Adapter authorization checking is disabled, and access to resources is not restricted. If PROTECTALL is active in RACF, AGH.RS must be defined.

If AGH.RS is defined, whether a client is granted access to a resource depends upon (1) its access authority for that resource and (2) the access authority of the process user (that is, of the user to whom the batch, CICS, or IMS environment belongs) for AGH.RS:

- If the process user has **no access authority** for AGH.RS, the client is denied access to all protected resources.
- If the process user has **READ**, **UPDATE**, or **CONTROL** access authority for AGH.RS, the client is granted access to the protected resource only if the client user has the requisite authorization for that resource. An access request that is denied is logged by RACF.
- If the process user has **ALTER** access authority for AGH.RS, the client can access all protected resources.

In a batch environment, the client has the same user ID as the process user; in an IMS or CICS environment, you can specify the client user ID via the AGHClient::setUserId method (for C++) or AGHClientSetUserId function (for C or COBOL).

Example of using authorization checking

```
⋮
//... JOB ...,USER=pro_user      [1]
⋮
//AGHEAPRO DD *
⋮
CorrespondentName=CN
⋮
AGH Client Program              [2]
⋮
status=SwCall(...,...);
⋮
AGHClientSetUserId("clt_user");   [3]
⋮
status=SwCall(...,...);
⋮
```

Figure 12. Example of using authorization checking in an OS/390 process

[1] The process user, who has the user ID **pro_user**, must have at least READ access to FACILITY(AGH.RS). The resource profile must contain the following entry:

```
FACILITY  Name    AGH.RS
          Access  READ
          Userid  pro_user
```

[2] The default client user is the process user. To access the correspondent name **CN**, the resource profile must contain the following entry:

```
AGH$CORN  Name    CN
          Access  READ
          Userid  pro_user
```

[3] The client user ID can be set explicitly via the AGHClient::setUserId method (for C++) or AGHClientSetUserId function (for C or COBOL). To be able to access the correspondent name **CN**, the client user, who has the user ID **clt_user**, must have at least READ access to the resource. The RACF resource profile must contain the following entry:

```
AGH$CORN  Name    CN
          Access  READ
          Userid  clt_user
```

Determining which RACF profile controls access to a protected resource

Access to protected resources is controlled by RACF profiles, which are located in RACF resource classes. The profiles that control access to correspondents have the same names as the correspondents themselves. The profiles that control access to DNs have names that are based on the DNs, but that are structured differently:

- A requestor, responder, or sign DN is a sequence of from 1 to 6 directory nodes (including a node-type identifier, for example cn= or o=) separated by commas. It specifies a path in an X.500 directory from leaf to root, for example:

```
cn=dept023,cn=branch03,o=bank01,o=swift
```

Blanks between nodes and case (whether characters are uppercase or lowercase) are ignored.

- A RACF profile for such a DN has a name that is comprised of a sequence of from 1 to 6 directory nodes (without the node-type identifier) separated by periods, in uppercase, and specifying the path in the X.500 directory from root to leaf. The profile name that corresponds to the DN in the example above would be:

```
SWIFT.BANK01.BRANCH03.DEPT023
```

Profile names can contain one or more asterisks (*) as wildcard characters that represents any string of characters. Profiles with such names are called *generic profiles*. If more than one profile applies to a single resource name (because of wildcard characters), the profile with the name that is most specific is used. For example, if there are profiles with the following names:

```
SWIFT.BANK01.*
SWIFT.BANK01.BRANCH*
SWIFT.BANK01.BRANCH03.DEPT02*
```

the resource `cn=dept023,cn=branch03,o=bank01,o=swift` would use the profile `SWIFT.BANK01.BRANCH03.DEPT02*`, because of the three profile names that apply, it is the most specific.

Chapter 2. Planning for and installing MQSeries Adapter

This chapter describes the things you need to consider before and while you install MQSeries Adapter.

Planning for MQSeries Adapter

Before you install MQSeries Adapter, check that your systems fulfill the hardware and software requirements, and that both of the following products are installed, configured, and available on the workstations that are to be connected to the SIPN:

- SWIFTAlliance Gateway Release 1.2.0
- SWIFTAlliance Gateway MQSeries Host Adapter Release 1.2.0

Hardware requirements

You need an ESA/390 processor (or a compatible processor) that can run OS/390 Version 2.6 or higher, and with enough processor storage to meet the combined requirements of the host operating system, the data communication (DC) system (CICS TS or IMS), and your application programs.

Software requirements

The base requirements are:

- OS/390 Version 2.6 (5647-A01)
- SMP/E Release 8 (5668-949)
- IBM Language Environment for OS/390 & VM Release 8 (5647-A01)
- MQSeries for OS/390 Version 2.6 (5655-A95), with the Client Attachment feature

You might want to use either of the following DC systems:

- CICS Transaction Server (CICS TS) for OS/390 Release 3 (5655-147)
- IMS/ESA Version 6 (5655-158)

Depending on the DC system, programming languages, and utilities you use, you might also require any of the following:

- C/C++ Compiler for OS/390 Version 2.6 (included in OS/390 Version 2.6)
- IBM COBOL for OS/390 & VM Version 2 (5648-A25)
- XML Toolkit for OS/390 (5655-D44); necessary if transferring SNL messages
- RACF Version 2.6 (5695-039)

Installing MQSeries Adapter

The information contained in the MQSeries Adapter program directory covers installation up to the point where the various product libraries have been stored on the host system. However, if you plan to use MQSeries Adapter in a CICS or IMS environment, or if you want to use the MQSeries Adapter authorization checking capability, there are some additional installation tasks that need to be carried out first.

© Copyright IBM Corp. 2001

Connect MQSeries Adapter to SAG

MQSeries Adapter uses the SAG MQSeries Host Adapter interface to connect a client or server to an SAG instance. How to set up such a connection is described in the *SWIFTAlliance Gateway MQHA Installation and Configuration Release 1.2.0*.

Copy DTD files to MVS

MQSeries Adapter uses the following document type definitions (DTDs) when retrieving DNs from SNL messages:

- Sw.dtd
- SwGbl.dtd
- SwSec.dtd
- SwInt.dtd

On an SAG workstation, these DTDs are located in either of the following locations:

- On a Windows NT® workstation, in the directory **C:\SWIFT\SagRls12\data\dtd**, where **C:\SWIFT\SagRls12** is the default installation directory (this might have been changed for your installation)
- On a UNIX® workstation, in the directory **/home/SWIFT/SagRls12/data/dtd**, where **/home/SWIFT/SagRls12** is the default installation directory (this might have been changed for your installation)

If you plan to transfer SNL messages, you must first copy these DTDs as binary files from a workstation on which SAG is installed to your MVS system into one of the following locations:

- If, in a client or server profile, the DTDFile parameter is specified, or if the run-time parameter POSIX(ON) is set, the DTDs must be located in the MVS hierarchical file system (HFS). In this case, the path and file names are case sensitive. Normally, the files are stored in the home directory of the process user; only if they are stored in a different directory do you need to specify the DTDFile parameter (see Table 1 on page 22).
- If POSIX(ON) is not set, the files must be stored as a local data set owned by the processing user. In this case, the data set names must consist of uppercase characters only. Each filename will have the form **uid.name.DTD**, where **uid** is the user ID of the process user and **name** is the name of the DTD. For example, if the user ID is AGH, the data set name of the first DTD is **AGH.SW.DTD**.

To avoid a DTD file mismatch, it is good practice to keep all SAGs at the same level.

CICS and IMS dependencies and restrictions

If you use MQSeries Adapter in a CICS or IMS environment, certain dependencies and restrictions apply.

CICS dependencies and restrictions

Before you can run the CICS versions of the samples, you must do the following:

1. Use the CICS system definition (CSD) utility DFHCSDUP to define resources for both OS/390 Language Environment and MQSeries. The CICS resource definitions for:
   - OS/390 Language Environment are located in member CEECCSD of the Language Environment library SCEESAMP. For more information on setting up the CICS environment for Language Environment, refer to the *Language Environment for OS/390 Customization* manual.

- CICS definitions for MQSeries are located in member CSQ4B100 of the MQSeries library SCSQPROC. For more information on setting up the CICS environment for MQSeries, refer to the ″MQ and CICS″ section of the *MQSeries for OS/390 System Management Guide*.

2. Add the necessary libraries to the CICS startup JCL. The DFHRPL and STEPLIB concatenations in your CICS startup JCL would then look like this:

```
⋮

//STEPLIB  DD  DSN=SYS1.CICS.SDFHAUTH,DISP=SHR
//         DD  DSN=SYS1.SCSQANLE,DISP=SHR   MQ IN THIS ORDER:1
//         DD  DSN=SYS1.SCSQAUTH,DISP=SHR     ..            2
//         DD  DSN=SYS1.SCEERUN,DISP=SHR    LE
//DFHRPL   DD  DSN=hlq.LOADLIB,DISP=SHR     private
//*  MQSeries Adapter uses the following 2 load libraries:
//         DD  DSN=hlq.SAGHLODC,DISP=SHR    CICS DLLs
//         DD  DSN=hlq.SAGHLOAD,DISP=SHR    DLLs
//         DD  DSN=SYS1.CICS.SDFHLOAD,DISP=SHR
//         DD  DSN=SYS1.SCSQANLE,DISP=SHR   MQ IN THIS ORDER:1
//         DD  DSN=SYS1.SCSQCICS,DISP=SHR     ..            2
//         DD  DSN=SYS1.SCSQAUTH,DISP=SHR     ..            3
//         DD  DSN=SYS1.SCEECICS,DISP=SHR   LE IN THIS ORDER:1
//         DD  DSN=SYS1.SCEERUN,DISP=SHR      ..            2
//DFHCSD   DD  DSN=hlq.DFHCSD,DISP=SHR      CICS sys.defs.
⋮
```

   In the DFHRPL concatenation, the SAGHLODC library must precede the SAGHLOAD library. The SAGHLODC library contains the DLLs for the client and server functions that are to be used under CICS, which in turn use the MQ CICS DLLs IMQS23DC and IMQB23DC. The connection from MQSeries to CICS must have been established before the DLLs can be used.

If you use MQSeries Adapter in a CICS environment, the following apply:

- The MQSeries Adapter server and client functions use OS/390 functions such as the system logger, RACF interface, and QSAM file access. Therefore, transactions and programs using the client and server functions must be defined with EXECKEY(CICS).
- The user ID of the user issuing the request (the *client user*) should be set using the AGHClient::setUserId method (for C++) or AGHClientSetUserId function (for C or COBOL); otherwise the user ID under which CICS runs will be used for authorization checking. The sample program AGHCAAEC shows how to extract the user ID and set the user ID.
- Because overflows of trace and message logging data sets cannot be detected, it is recommended that, instead of using data sets, you route trace output to SYSOUT, and route log records to a system logger stream.

In a CICS environment, the following restrictions apply:

- XML Toolkit for OS/390 does not run as a CICS transaction. As a result, MQSeries Adapter cannot transfer SNL messages in a CICS environment.
- You cannot use the MQSeries Adapter server console interface. You must instead use standard CICS methods (such as CICS temporary storage queues) to communicate with a running server transaction.

To use a DLL in a CICS environment:

- Define the DLL as PROGRAM in the CSD, with LANGUAGE(LE370).
- Concatenate the load library containing the DLL to DFHRPL.
- Use OS/390 LE run-time libraries.
- Call the DLL using the call interface for your programming language (not EXEC CICS LINK).

For COBOL, the DLL cannot use the DISPLAY functions. For more information about this function, refer to “Programming Considerations for CICS” in the *OS/390 COBOL Programming Guide*.

IMS dependencies and restrictions

The IMS client and server samples run as IMS message processing program (MPP) transactions. Before you can run the IMS versions of the MQSeries Adapter samples, ask your IMS administrator to define IMS transactions, PSBs, and ACBs for the following:

AGHCAAA7 InterAct client sample (for C)

AGHCAAA9 InterAct client sample (for COBOL)

AGHCAAEI InterAct server sample (for C++)

Examples of IMS transaction definitions are shown in the *SWIFTAlliance Gateway MQHA Application Programming Guide Release 1.2.0*.

The JCL to start an IMS MPP for an MQSeries Adapter client or server transaction must include the MQSeries Adapter load library in the STEPLIB, as shown in the following example:

```
⋮
//STEPLIB  DD  DSN=hlq.LOADLIB,DISP=SHR      private
//         DD  DSN=hlq.SAGHLOAD,DISP=SHR     DLLs
//         DD  DSN=SYS1.IMS.RESLIB,DISP=SHR
//         DD  DSN=SYS1.SCSQLOAD,DISP=SHR
//DFSESL   DD  DSN=SYS1.IMS.RESLIB,DISP=SHR
           DD  DSN=SYS1.SCSQLOAD,DISP=SHR
⋮
```

Implementing authorization checking

The steps you need to undertake to enable your ESM to provide MQSeries Adapter authorization checking services are described in “Appendix C. Considerations when implementing authorization checking” on page 113.

Chapter 3. Creating client and server profiles

The parameters for each client and server is contained in the file named in the DD statement AGHEAPRO. This file contains the profiles for the clients and servers (one profile for each). Each profile consists of a label that identifies the client or server, followed by a series of parameters of the form **keyword=value**. The label can be up to 32 bytes long; the last character must be a colon (:). The keywords are case insensitive. The parameter values must not contain any blanks, because the characters following the first blank are interpreted as a comment. Also, lines beginning with a pound sign (#) are interpreted as comments. An example is shown in Figure 13.

```
SERVER01:                                      Server01 RequestHandler
  MQMName       = CSQ1                         Queue manager name
  MQMRequestQ   = AGH.SERVER.REQUEST           Request queue name
  MQMReports    = COA,COD                      MQSeries reports (debug)
  CorrespondentName = MQBasicServer            Message partner name
  TraceLevel    = 30                           Trace level (Debug)
  TimeOut       = 60                           Time out in seconds
  Expiry        = 60                           Expiry in seconds
  Log           = Y                            Audit log Y or N
  LogDDName     = AGHEALOG                     DD name log data set
  LogStreamName = AGH.LOG                      System logger stream
  LogFinInst    = BANKNAME                     Institution (log record)
  LogFinBrch    = IT-Test                      Branch (log record)
  LogFinDept    = Server                       Department (log record)
  SecSVCNum     = 215                          SVC number for Security
  CEEPrefix     = SYS1                         Prefix of ICONV tables
  DTDFile       = /u/agh/dtd/DTD               DTDs
  Console       = Y                            Console Interface Y or N
  Convert       = Y                            Conversion Y (basic mode)
```

Figure 13. Example of a file containing client and server profiles (Part 1 of 3)

```
SERVER02:                                      Server02 ClientHub
  MQMName       = CSQ1                         Queue manager name
  MQMRequestQ   = AGH.CLIENTHUB.REQUEST        Request queue name
  MQMRequestFileQ = AGH.CLIENTHUB.FILE         File Queue for ClientHub
  MQMReports    = COA,COD                      MQSeries reports (debug)
  TraceLevel    = 30                           Trace level (Debug)
  TimeOut       = 60                           Time out in seconds
  Expiry        = 60                           Expiry in seconds
  Log           = Y                            Audit log Y or N
# LogDDName     = AGHEALOG                     DD name log data set
  LogStreamName = AGH.LOG                      System logger stream
  LogFinInst    = BANKNAME                     Institution (log record)
  LogFinBrch    = IT-Test                      Branch (log record)
  LogFinDept    = ClientHub                    Department (log record)
  SecSVCNum     = 215                          SVC number for Security
  CEEPrefix     = SYS1                         Prefix of ICONV tables
  DTDFile       = /u/agh/dtd/DTD               DTDs
  Console       = Y                            Console Interface Y or N
  ClientName    = CLIENT01                     Client Name for ClientHub
```

Figure 13. Example of a file containing client and server profiles (Part 2 of 3)

© Copyright IBM Corp. 2001

```
CLIENT01:                                     Client01 parameter
  MQMName       = CSQ1                        Queue manager name
  MQMRequestQ   = AGH.CLIENT.REQUEST          Request queue name
  MQMReplyToQ   = AGH.CLIENT.REPLYTO          ReplyTo queue name
  MQMFileQ      = AGH.CLIENT.FILE             LFT File queue name
  MQMReports    = COA,COD                     MQSeries reports (debug)
  CorrespondentName = MQBasicClient           Message partner name
  TraceLevel    = 30                          Trace level (Debug)
  TimeOut       = 60                          Time out in seconds
  Expiry        = 60                          Expiry in seconds
  Log           = Y                           Audit log Y or N
  LogDDName     = AGHEALOG                    DD name log data set
  LogStreamName = AGH.LOG                     System logger stream
  LogFinInst    = BANKNAME                    Institution (log record)
  LogFinBrch    = IT-Test                     Branch (log record)
  LogFinDept    = Client                      Department (log record)
  SecSVCNum     = 215                         SVC number for Security
  CEEPrefix     = SYS1                        Prefix of ICONV tables
  DTDFile       = /u/agh/dtd/DTD              DTDs
```

Figure 13. Example of a file containing client and server profiles (Part 3 of 3)

The parameters and the values they can have are described in the following tables. The lengths shown are in characters.

Table 1. General Parameters

Parameter	Description	Length	Client	Server
CCSID	The coded character set identifier (CCSID) is the number of the code page to be used when converting basic or EBCDIC SNL messages (see “Data conversion” on page 11).	4	default= 1047	default= 1047
CEEPrefix	MQSeries Adapter uses the ICONV utility to convert data from the IBM-1047 code page to UTF-8 and vice versa. This utility stores its conversion tables in the data sets ***hlq*.SCEEUMAP** and ***hlq*.SCEEUTBL**, where ***hlq*** is a high-level qualifier. To enable MQSeries Adapter to find these data sets, specify this high-level qualifier with this parameter.	44	optional; no default	optional; no default
ClientName	The client hub uses this parameter to indicate which set of client profile parameters are to be used for the requests it processes. The name specified must correspond to the label of a client profile.	32	not applicable	default= CLIENT01
Console	This parameter determines whether to activate the console interface for starting and stopping a client hub or server request handler. Specify **Y** to activate it; any other character not activate it.	1	not applicable	default=N
Convert	This parameter determines whether the server is to use the MQSeries conversion facility to convert the messages it receives from SAG and, if so, which type of conversion to use: **Y** For basic messages **X** For SNL messages encoded in EBCDIC format **N** For SNL messages encoded in UTF-8 format For more information, see “Data conversion” on page 11.	1	not applicable	default=N
CorrespondentName	The name of the correspondent (message partner): • For a client, this name is passed to SAG and used as a key to find the correspondent’s SAG profile. • For a server, this name is only used to check whether the process user is authorized to send responses.	24	mandatory	optional; no default

Table 1. General Parameters (continued)

Parameter	Description	Length	Client	Server
DTDFile	This parameter is needed only if MQSeries Adapter is to process SNL messages, and if the DTDs it needs to do this are located in a directory other than the home directory of the process user. For more information about DTDs, see "Copy DTD files to MVS" on page 18. This parameter specifies the directory in which the DTDs are stored, plus a dummy file name of your choosing. For example, if the DTDs are stored in the directory **/u/agh/dtd**, specify **DTDFile=/u/agh/dtd/DTD**, where **DTD** is a dummy file name. This file name is ignored, except when an attempt to access a DTD file results in an error, in which case this file name is shown in the resulting error message.	48	optional; no default	optional; no default
Expiry	Maximum amount of time (in seconds) that a message (request or response) can remain in a queue before it will no longer be retrieved. An expired message remains in a queue until the next MQSeries Browse or Read command, at which time it is purged. If this parameter is not specified, or if its value is 0, there is no limit on the amount of time messages can remain in queues.	9	default=0	default=0
SecSVCNum	The security SVC number. Specify this if you access resources using a user ID other than that of the process user (that is, the user to whom the batch, CICS, or IMS environment belongs). This is possible only when using the client hub, or when running under IMS or CICS. The SVC number must be in the range from 200 to 255.	3	optional; no default	optional; no default

TimeOut	One of the following:
	• For a client, the number of seconds that MQSeries Adapter will wait for a response from SwCall or SwAWait
	• For a client hub or server request handler, the number of seconds that MQSeries Adapter will wait before shutting down the client hub or server request handler after either:
	  – Its request queue is empty (if Console=N)
	  – It was stopped via the MODIFY command with the parameter APPL=SHUT (if Console=Y)
	• For a server, the number of seconds the server will wait for a retrieve function to complete before continuing its processing
	
	If this parameter is not specified, or if its value is 0, there is no time limit.
		9	default=0	default=0

Table 2. MQSeries Parameters

Parameter	Description	Length	Client	Server
MQMFileQ	Name of the SAG file queue. This is the queue into which the AGHLFTCmd puts file data that is to be moved to the system on which SAG runs (usually in preparation for a FileAct transfer). If the application uses a remote queue, this is the local definition of that remote queue.	48	optional; no default	not applicable

Table 2. MQSeries Parameters (continued)

Parameter	Description	Length	Client	Server
MQMName	Name of the MQSeries queue manager that manages the queue to which the request is to be sent.	48	optional; no default	optional; no default
MQMReports	Which types of reports MQSeries is to create: **COA** Confirm on arrival only **COD** Confirm on delivery only **COA,COD** Both confirm on arrival and confirm on delivery	3 or 7	optional; no default	optional; no default
MQMReplyToQ	Name of the reply-to queue. This is the local queue in which SAG stores responses.	48	mandatory	not applicable
MQMRequestFileQ	Name of the local MQSeries Adapter queue in which back-office applications can store file data that the client hub is to transfer using LFT commands.	48	not applicable	optional; no default
MQMRequestQ	For a client or server, the name of the queue in which the request is to be placed at the client or server site. If the client or server uses a remote queue, this is the local definition of that remote queue. For a client hub, the name of the central request queue; that is, the queue from which it retrieves requests.	48	mandatory	mandatory

Table 3. Message logging and tracing parameters

Parameter	Description	Length	Client	Server
Log	Whether MQSeries Adapter is to log messages. Specify **Y** to activate message logging; **N** to deactivate message logging. If you activate message logging, you must also specify either or both LogDDName or LogStreamName. See "Message logging" on page 87 for more information.	1	default=N	default=N
LogDDName	DD name of the sequential log data set to which log records are to be written. The log entry is written to the sequential data set allocated to the specified DD name.	8	optional; no default	optional; no default
LogFinBrch	User-defined information about the user's branch, for example its name. The value of this parameter is included in the header of each log record, and can be used for further processing of the log records.	12	optional; no default	optional; no default
LogFinDept	User-defined information about the department, for example its name. The value of this parameter is included in the header of each log record, and can be used for further processing of the log records.	12	optional; no default	optional; no default
LogFinInst	User-defined information about the financial institution, for example its bank identifier code (BIC). The value of this parameter is included in the header of each log record, and can be used for further processing of the log records.	12	optional; no default	optional; no default
LogStreamName	Name of the system logger stream to which log records are to be written.	48	optional; no default	optional; no default
TraceLevel	The trace level (see "Tracing" on page 90 for more information).	2	default=0	default=0

Chapter 4. Writing MQSeries Adapter clients and servers, and SwCallback functions

After you have installed MQSeries Adapter (see “Installing MQSeries Adapter” on page 17) and have defined a profile for each of the clients and servers you intend to use (see “Chapter 3. Creating client and server profiles” on page 21), you can begin writing the clients and servers:

- If coding in C++, use the classes and methods described in “Chapter 5. MQSeries Adapter C++ classes and methods” on page 31.
- If coding in C or COBOL, use the functions described in “Chapter 6. MQSeries Adapter functions for C and COBOL” on page 49.

If, instead of writing your own server, you elect to use the server request handler provided by MQSeries Adapter, you must provide both an SwCallback function to process the retrieved requests, and an AppXmlBufferFree function to release the response buffer that the SwCallback function allocates. For more information about these functions, see “Chapter 7. User-written functions” on page 83.

Clients and servers use the data structures described in the following sections. MQSeries Adapter provides samples of clients and servers, as well as of client hubs, server request handlers, and SwCallback functions. These are described in “Sample programs” on page 29.

Data structures

The data structures described in this section comprise the interface that clients use to exchange data with MQSeries Adapter. They are located in the sample library:

- For C and C++, in the header file with the name AGHCCTYP
- For COBOL, in the copybook AGHEACB1

AGHResponse

AGHResponse is used by the functions SwACall and SwAWait, and is shown in Figure 14 and Figure 15.

```
typedef struct
{
   CHAR     *Resp;              /* Pointer to the response       */
   CHAR     MsgId[24];          /* Message id passed from MQPut   */
   LONG     lCodePageId;        /* Code page for msg from SwACall */
} AGHResponse;
```

Figure 14. AGHResponse data structure (C and C++)

© Copyright IBM Corp. 2001

```
     01   AGHResponse.
    *          Pointer to the response
          02   Resp                         POINTER.
    *          Message id passed from MQPut
          02   MsgId                        PIC X(24).
    *          Code page for msg from SWACall
          02   lCodePageId                  PIC S9(9) BINARY.
```

Figure 15. AGHResponse data structure (COBOL)

This data structure contains the following fields:

Resp
In this field MQSeries Adapter stores the pointer to the response buffer allocated by MQSeries Adapter. In case of an error, the response buffer contains the error response.

MsgId
This information is used for the SwAWait to identify the response message in the reply-to queue. This field is filled by MQSeries Adapter during the SwACall.

lCodePageId
This information is used to make sure that the response from SwAWait is passed back using the same code page as was used to pass the request to SAG. This field is filled by MQSeries Adapter during the SwACall.

AGHLFTCmdParm

Use AGHLFTCmdParm to pass an LFT command to MQSeries Adapter. MQSeries Adapter uses AGHLFTCmdParm to pass back to you its return code and response.

The AGHLFTCmdParm data structure is shown in Figure 16 and Figure 17.

```
typedef struct
{
  CHAR cLFTVersion[3];          /* Version of LFT interface     */
  CHAR cLFTCmd[7];              /* Command or return code       */
  CHAR cLFTText[1];             /* Begin command or return text */
} AGHLFTCmdParm;
```

Figure 16. AGHLFTCmdParm data structure (C and C++)

```
     01   AGHLFTCmdParm.
    *          Version of LFT interface
          02   cLFTVersion                  PIC X(3).
    *          Command or return code
          02   cLFTCmd                      PIC X(7).
    *          Command or return text
          02   cLFTText                     PIC X(512).
```

Figure 17. AGHLFTCmdParm data structure (COBOL)

This data structure contains the following fields:

cLFTVersion
Whether issuing a command or receiving a response, this field contains a string indicating the version of the LFT interface provided by SAG.

cLFTCmd
When issuing a command, this field contains one of the following (padded with trailing blanks):

PUT Put a file from an OS/390 system onto an SAG workstation

GET Get a file from an SAG workstation and move it to an OS/390 system

LIST List the files on an SAG workstation that meet the specified criteria

DELETE Delete one or more files from an SAG workstation

When receiving a response, this is the SAG return code padded with trailing blanks.

cLFTText
When issuing a command, this field contains a null-terminated string specifying the complete relative file name (for PUT, GET, and DELETE commands) or the relative path (for a LIST command).

When receiving a response:

- In case of an error, this field contains an error description.
- In case of a successful PUT command, this field contains the fully-qualified file name.
- In case of a successful LIST command, this field contains the full path followed by the directory entries. The path and entries are separated by EBCDIC carriage-return line-feed (CRLF) characters (X'0D25').

For a more detailed description of these fields, see the description of LFT commands in the *SWIFTAlliance Gateway MQHA Application Programming Guide Release 1.2.0*.

AGHFileHeader

AGHFileHeader is used for LFT commands:

- When you issue a PUT command, you use AGHFileHeader to pass to MQSeries Adapter:
  - The length of the file data in the field lDataLength
  - A pointer to the buffer containing the file data in the field pData
- MQSeries Adapter uses AGHFileHeader to pass to you, in response to a:
  - PUT command, the name of the reply-to queue (szReplyTQ), and of the queue manager to which the reply-to queue belongs (szToMQM). You might require this data for subsequent processing, for example to ensure that a subsequent FileAct request is sent to the correct SAG instance (the one running on the system where the file is stored).
  - GET command, the length of the file data and a pointer to the buffer containing the file data (lDataLength and pData).

The AGHFileHeader data structure is shown in Figure 18 and Figure 19.

```
typedef struct
{
   LONG     lDataLength;        /* Length of file data          */
   CHAR     *pData;             /* Pointer to the file data     */
   CHAR     lFileOpt;           /* Option for future use        */
   CHAR     szToMQM[49];        /* Reply to queue manager       */
   CHAR     szReplyTQ[49];      /* Reply to queue name          */
} AGHFileHeader;
```

Figure 18. AGHFileHeader data structure (C and C++)

```
       01   AGHFileHeader.
      *          Length of file data
            02   lDataLength                 PIC S9(9) BINARY.
      *          Pointer to the file data
            02   pData                       POINTER.
      *          Option for future use
            02   lFileOpt                    PIC X(1).
      *          Reply-to queue manager
            02   szToMQM                     PIC X(49).
      *          Reply-to queue name
            02   szReplyTQ                   PIC X(49).
```

Figure 19. AGHFileHeader data structure (COBOL)

XML messages

If a request is an SNL message, its response will be an XML message. If the response contains an error message, it will have a format similar to that shown in Figure 20 (if issued by the client) or Figure 21 on page 29 (if issued by the server).

```
<?xml version= "1.0" encoding= "IBM-1047" ?>
<!DOCTYPE SwInt:ExchangeResponse SYSTEM "Sw.dtd" >
<SwInt:ExchangeResponse>
 <SwGbl:Status>
  <SwGbl:StatusAttributes>
   <SwGbl:Severity>Transient</SwGbl:Severity>
   <SwGbl:Code>AGH.Client.ErrorMessage</SwGbl:Code>
   <SwGbl:Parameter>AGHxxxxE Error message.</SwGbl:Parameter>
   <SwGbl:Text>For more information see client JOB output or client tracefile.</SwGbl:Text>
   <SwGbl:Action></SwGbl:Action>
  </SwGbl:StatusAttributes>
 </SwGbl:Status>
</SwInt:ExchangeResponse>
```

Figure 20. Example of an XML message from a client

```
<?xml version="1.0" encoding="IBM-1047" ?>
<!DOCTYPE SwInt:HandleResponse SYSTEM "Sw.dtd" >
<SwInt:HandleResponse>
 <SwInt:Response>
  <SwInt:ResponsePayload>
   <SwGlb:Status>
    <SwGlb:StatusAttributes>
     <SwGlb:Severity>Transient</SwGlb:Severity>
     <SwGlb:Code>AGHServerError</SwGlb:Code>
     <SwGlb:Parameter>AGHxxxxE Error message.</SwGlb:Parameter>
    </SwGlb:StatusAttributes>
   </SwGlb:Status>
  </SwInt:ResponsePayload>
 </SwInt:Response>
</SwInt:HandleResponse>
```

Figure 21. Example of an XML message from a server

For more information about SNL messages, see the *SWIFTAlliance Gateway Interface Specification Release 1.2.0*.

Sample programs

MQSeries Adapter provides the following sample application programs for the following programming languages (note that some programs apply to more than one language or environment):

Table 4. Sample application programs

<table>
<tr><th colspan="2" rowspan="2">Sample</th><th colspan="3">Programming Language</th><th rowspan="2">Description</th></tr>
<tr><th>C</th><th>C++</th><th>COBOL</th></tr>
<tr><td rowspan="3">InterAct client</td><td>batch</td><td>AGHCAAA1</td><td>AGHCAAA2</td><td>AGHCAAA3</td><td rowspan="3">Initiates a synchronous InterAct message exchange; that is, uses the function SwCall to send a message to a partner and receive the partner's response.</td></tr>
<tr><td>CICS</td><td>AGHCAAA4</td><td>AGHCAAA4</td><td>AGHCAAA6</td></tr>
<tr><td>IMS</td><td>AGHCAAA7</td><td>AGHCAAA7</td><td>AGHCAAA9</td></tr>
<tr><td rowspan="3">InterAct server SwCallback function</td><td>batch</td><td>AGHCAAB1</td><td>AGHCAAB1</td><td>AGHCAAB3
AGHCAAB4</td><td rowspan="3">Provides the SwCallback function as a DLL. The server passes a message to this program, and this program returns a response to the server.

For COBOL, a second sample is also provided. It uses the AppXmlBufferFree function (see page 85) to release the response buffer allocated by the SwCallback function.</td></tr>
<tr><td>CICS</td><td>AGHCAAB1</td><td>AGHCAAB1</td><td>AGHCAAB6
AGHCAAB7</td></tr>
<tr><td>IMS</td><td>AGHCAAB1</td><td>AGHCAAB1</td><td>AGHCAAB3</td></tr>
<tr><td rowspan="3">InterAct server</td><td>batch</td><td></td><td>AGHCAAEB</td><td></td><td rowspan="3">Retrieves requests from the request queue and for each places a response into the reply-to queue. The programs for CICS and IMS can be triggered by MQSeries.</td></tr>
<tr><td>CICS</td><td></td><td>AGHCAAEC</td><td></td></tr>
<tr><td>IMS</td><td></td><td>AGHCAAEI</td><td></td></tr>
<tr><td>LFT client</td><td>batch</td><td></td><td>AGHCAAF1</td><td></td><td>A batch program that demonstrates the use of the AGHLFTCmd interface for local file transfer (LFT). Compared to the client hub sample program for LFT (AGHCAAF2), this program does not use the MQSeries queues that serve as the interface to the client hub.</td></tr>
</table>

Table 4. Sample application programs (continued)

Sample		Programming Language			Description
		C	C++	COBOL	
LFT client for client hub	batch		AGHCAAF2		A batch program that demonstrates the use of the client hub for a local file transfer (LFT). The interface to the client hub consists of several MQSeries queues. This program: • Reads or writes a file being transferred to or from one MQSeries queue • Writes the corresponding SAG LFT command to a second MQSeries queue • Reads the response to the LFT command from a third MQSeries queue
Client hub	batch	AGHCAAS2	AGHCAAS1		A program that starts an MQSeries Adapter client hub.
Server request handler	batch	AGHCAAS2	AGHCAAS1		A program that starts an MQSeries Adapter

The sample programs are located in the following libraries:
- For C and C++, in the library with the low-level qualifier SAGHSAMC
- For COBOL, in the library with the low-level qualifier SAGHSAMB

Chapter 5. MQSeries Adapter C++ classes and methods

MQSeries Adapter provides the following C++ classes:

- **AGHClient**, which is used to create client objects, is contained in the DLL **AGHCADLC**.
- **AGHServer**, which is used to create server objects, is contained in the DLL **AGHCADLS**.

The condition codes issued by clients and servers are described in “Condition codes” on page 104. To see the accompanying error message, check the trace or retrieve the error message using the getErrorMessage method. If you require more information, consider setting a higher trace level as described in “Tracing” on page 90.

AGHClient

The class AGHClient encapsulates a client object. The client class definitions are in the header file AGHCASWD.

Constructors

When you create your first client object, an internal logger service object is created that writes trace and log entries. If you create a second client object, and if you want both to use the same logging and trace data sets, you must pass the logger service object from the first client object. To get the logger service object, use the getLoggerObj method as shown in the example below.

Syntax:

```
AGHClient(CHAR * pszClientName, AGHLoggerService * pLog = NULL);
AGHClient(AGHLoggerService * pLog = NULL);
```

where:

pszClientName **(input)**
A pointer to a null-terminated string containing the name of the client object. If you do not specify a name, you can specify one later via the setClientName method (see “AGHClient::setClientName” on page 35). If you don’t specify a name, the client name is set to CLIENT01 during the first call that initiates a message transfer. Whether you specify a name or use the default name, your profile data set must contain a profile for a client with that name (see “Chapter 3. Creating client and server profiles” on page 21).

pLog **(input)**
A pointer to the logger object of a previously created AGHClient object. If this is not specified, a new logger object is created for the client.

Coding example:

```
#include <aghcaswd.hpp>
⋮

pClient01 = new AGHClient();
```

© Copyright IBM Corp. 2001

```
pClient02 = new AGHClient("CLIENT02", pClient01->getLoggerObj());
⋮
```

Methods

AGHClient::AGHFCall

This method initiates a FileAct transfer for a file that is already on an SAG workstation. This file might have been moved to that system by an AGHLFTCmd call, but not necessarily. To make sure that the request is sent to the same instance of SAG that processed the corresponding AGHLFTCmd, the same AGHFileHeader structure that was used by that AGHLFTCmd must be used by this method. For more information about local file transfers, see:

- "AGHLFTCmdParm" on page 26
- "AGHFileHeader" on page 27
- "AGHClient::AGHLFTCmd" on page 33

Syntax:

```
SwStatus AGHFCall(AGHSTRING req, AGHPSTRING pResp, AGHFileHeader* fhead);
```

where:

req **(input)**
: The name of the file stored on the SAG workstation, and that is to be transferred via the SIPN to a remote location (that is, for which a FileAct is to be started). For information on how to specify such names, refer to the *SWIFTAlliance Gateway MQHA Application Programming Guide Release 1.2.0*.

pResp **(output)**
: A pointer to the address of the response buffer. MQSeries Adapter allocates memory for the response buffer; the program must free this memory after it is no longer needed.

fhead **(input)**
: A pointer to the AGHFileHeader structure, which contains the name of the queue manager and the name of the target queue to which this request must be sent (see "AGHFileHeader" on page 27).

Return values: SwOperationSucceed or SwOperationFailed.

Coding example:

```
⋮

#include <iostream.h>
#include <aghcaswd.hpp>
⋮

AGHClient *    pClient01 = NULL;
AGHFileHeader  file01;
AGHCHAR        szReqBuffer[512];
AGHSTRING      Response;
SwStatus       status;

pClient01 = new AGHClient();

memset(szReqBuffer, '\0', sizeof(szReqBuffer));
strcpy(szReqBuffer, "x:/path/file.ext");

status = pClient01->AGHFCall(szReqBuffer, &Response, &file01);
if(status == SwOperationSucceed)
{
```

```
   /* ok */
   cout << "AGHFCall response: "<<Response <<endl;
}
else {
   /* failed */
   cout << "AGHFCall error response: "<<Response <<endl;
}

pClient01->SwXmlBufferFree(Response);
delete pClient01;
```

AGHClient::AGHLFTCmd

This method is used to put files onto an SAG workstation (for example, in preparation for a FileAct transfer), or to get, list, or delete such files.

Syntax:

```
SwStatus AGHLFTCmd(AGHSTRING req, AGHPSTRING pResp, AGHFileHeader* fhead);
```

where:

req **(input)**
: A pointer to the local file transfer (LFT) command to be issued. These commands are described in "AGHLFTCmdParm" on page 26.

pResp **(output)**
: A pointer to an area containing the address of another area where the response is to be stored (the response buffer). The contents of this buffer can be read using the AGHLFTCmdParm data structure (see "AGHLFTCmdParm" on page 26). MQSeries Adapter allocates memory for the response buffer; the program must free this memory after it is no longer needed.

fhead **(input/output)**
: The address of the data structure containing the length and address of the data to be put (see "AGHFileHeader" on page 27). If a PUT command is issued in preparation for a FileAct request (AGHFCall), information needed by the FileAct request (for example, the names of the queue manager and the request queue to be used) is returned in this data structure.

Return values: SwOperationSucceed or SwOperationFailed.

Coding example:

```
⋮

#include <stdio.h>
#include <aghcaswd.hpp>
⋮

AGHClient *     pClient01 = NULL;
AGHFileHeader   file01;
AGHLFTCmdParm * pParm01;
AGHCHAR         szReqBuffer[512];
AGHSTRING       Response;
CHAR szFileBuffer[]="Test file Test file Test file Test file Test file";
SwStatus        status;

file01.lDataLength = sizeof(szFileBuffer);
file01.pData       = szFileBuffer;
pParm01 = (AGHLFTCmdParm*)szReqBuffer;
pClient01 = new AGHClient();
```

```
memset(szReqBuffer, ' ', sizeof(szReqBuffer));
memcpy(pParm01->cLFTVersion, AGHLFTVERSION,
                             sizeof(pParm01->cLFTVersion));
memcpy(pParm01->cLFTCmd, "PUT", 3);
strcpy(pParm01->cLFTText, "x:/path/file.ext");

status = pClient01->AGHLFTCmd(szReqBuffer, &Response, &file01);
if(status == SwOperationSucceed)
{
   /* ok */
   cout << "AGHLFTCmd response: "<<Response <<endl;
}
else {
   /* failed */
   cout << "AGHLFTCmd error response: "<<Response <<endl;
}

pClient01->SwXmlBufferFree(Response);
delete pClient01;
```

AGHClient::getConditionCode

MQSeries Adapter saves the last condition code that it issues. A client can use this method to retrieve this condition code. Possible values are described in Table 10 on page 104.

Syntax:

```
LONG  getConditionCode();
```

Coding example:

```
⋮
#include <aghcaswd.hpp>
⋮
AGHClient * pClient01 = NULL;
LONG       lCCode;
⋮
pClient01 = new AGHClient();
⋮
lCCode = pClient01->getConditionCode();
⋮
delete pClient01;
```

AGHClient::getErrorMessage

Use this method to retrieve a pointer to the error message that MQSeries Adapter last issued. The error messages are described in “Appendix A. Messages and codes” on page 95. The maximum size of the error message buffer is 256 bytes.

Syntax:

```
CHAR * getErrorMessage();
```

Coding example:

```
⋮
#include <aghcaswd.hpp>
⋮
AGHClient * pClient01 = NULL;
```

```
CHAR *      pszErrorMessage;
⋮

pClient01 = new AGHClient();
⋮

pszErrorMessage = pClient01->getErrorMessage();
⋮

delete pClient01;
```

AGHClient::getLoggerObj

Returns a pointer to the logger object of the client instance. A single logger object can be shared by several clients.

Syntax:

```
AGHLoggerObject * getLoggerObj();
```

For a coding example, see "Constructors" on page 31.

AGHClient::setClientName

Use this method to overwrite the label for the profile entry that is used to initialize the MQSeries Adapter client. If you overwrite the default name CLIENT01, you must do so as the first call to MQSeries Adapter.

Syntax:

```
LONG setClientName(CHAR * pszClientName);
```

where:

pszClientName **(input)**
: A pointer to a null-terminated string containing the client name. This name can be at most 32 characters long.

Return values: AGHOK (0) if successful; AGHERRORCLIENT (10) if length exceeds 32 characters.

Coding example:

```
⋮

#include <aghcaswd.hpp>
⋮

AGHClient* pClient01 = NULL;
CHAR       szClientName[] = "CLIENT02";
LONG       lCCode;
⋮

pClient01 = new AGHClient();
⋮

lCCode = pClient01->setClientName(szClientName);
if(lCCode == AGHOK)
{
  /* ok */
}
else
{
  /* failed */
}
```

```
⋮
delete pClient01;
```

AGHClient::setUserId

Use this method to set the user ID for which access authority is to be checked (see "Authorization checking" on page 13). If this method is not used, the user ID of the process user is assumed as a default. This method can only be called when running in CICS or IMS environment. Clients that process messages created by a CICS or IMS application should set the client user ID to the process user ID of the corresponding CICS or IMS environment.

Syntax:

```
LONG setUserId(CHAR * pszUid);
```

where:

pszUid **(input)**
: A pointer to a null-terminated string that is the user ID that is to be used for all subsequent authorization checking. This string can be at most 8 characters long.

Return values: AGHOK (0) if successful; AGHERRORSECURITY (16) if the length was greater than 8 characters, or if this method was called from a batch environment

Coding example:

```
⋮
#include <aghcaswd.hpp>
⋮
AGHClient* pClient01 = NULL;
CHAR       szUserId[]="USERID";
⋮
pClient01 = new AGHClient();
⋮
lCCode = pClient01->setUserId(szUserId);
if(lCCode == AGHOK)
{
  /* ok */
}
else {
  /* failed */
}
⋮
delete pClient01;
```

AGHClient::SwACall

This method is used to pass a request to MQSeries Adapter that is to be treated asynchronously; that is, control is to be passed back to the client immediately after the request has been added to the request queue. To retrieve the response associated with a request submitted using this call, use the SwAWait method (see "SwAWait—Retrieve a response from an asynchronous call" on page 77).

Syntax:

```
SwStatus SwACall(AGHSTRING req, AGHResponse* pAsyncResp);
```

where:

req **(input)**
: The address of the area containing the message to be sent (the request buffer). This message can be a basic or an SNL message.

pAsyncResp **(output)**
: The address of the AGHResponse structure where the message ID and code page information from SwACall is stored (see “Data structures” on page 25). In case of an error, this structure contains a pointer to the error buffer. MQSeries Adapter allocates memory for the response buffer; the program must free this memory after it is no longer needed.

Return values: SwOperationSucceed or SwOperationFailed.

Coding example:

```
⋮

#include <iostream.h>
#include <aghcaswd.hpp>
⋮

AGHClient   pClient01 = NULL;
AGHResponse AsyncResp;
SwStatus    status;
AGHCHAR     szMsgBuffer[] = "Hello World from client";

pClient01 = new AGHClient();
status = pClient01->SwACall(szMsgBuffer, &AsyncResp);
if(status == SwOperationSucceed)
{
   /* ok */
}
else
{
   /* failed */
   cout << "SwACall error response: "<<AsyncResp.Resp <<endl;
}
pClient01->SwXmlBufferFree(AsyncResp.Resp);
⋮

delete pClient01;
```

AGHClient::SwAWait

This method retrieves the response that resulted from a previously issued SwACall. This method uses the pointer to the area containing the address of the response area that was used by SwACall.

Syntax:

```
SwStatus SwAWait(AGHResponse* pAsyncResp);
```

where:

pAsyncResp **(input/output)**
: A pointer to the AGHResponse structure that contains the address of the response from SwACall or, in case of an error, the address of the error buffer. MQSeries Adapter allocates memory for the response buffer; the program must free this memory after it is no longer needed.

Return values: SwOperationSucceed or SwOperationFailed.

Coding example:

```
⋮
#include <iostream.h>
#include <aghcaswd.hpp>
⋮
AGHClient   pClient01 = NULL;
AGHResponse AsyncResp;
SwStatus    status;
AGHCHAR     szMsgBuffer[] = "Hello World from client";
⋮
status = pClient01->SwACall(szMsgBuffer, &AsyncResp);
⋮
status = pClient01->SwAWait(&AsyncResp);
if(status == SwOperationSucceed)
{
   /* ok */
   cout << "SwAWait response: " << AsyncResp.Resp << endl;
}
else
{
   /* failed */
   cout << "SwAWait error response: " << AsyncResp.Resp << endl;
}
⋮
pClient01->SwXmlBufferFree(AsyncResp.Resp);
delete pClient01;
```

AGHClient::SwCall

This method is used to pass a request to MQSeries Adapter that is to be treated synchronously; that is, control is not to be passed back to the client until after SAG returns the response. It is compatible with the native SAG interface.

Syntax:

```
SwStatus SwCall(AGHSTRING req, AGHPSTRING pResp);
```

where:

req **(input)**
: The address of the area containing the message to be sent (the request buffer). This message can be a basic or an SNL message.

pResp **(input)**
: A pointer to an area containing the address of another area where the response is to be stored (the response buffer). MQSeries Adapter allocates memory for the response buffer; the program must free this memory after it is no longer needed.

Return values: SwOperationSucceed or SwOperationFailed.

Coding example:

```
⋮
#include <iostream.h>
#include <aghcaswd.hpp>
⋮
```

```
AGHClient pClient01 = NULL;
AGHSTRING Resp;
SwStatus  status;
AGHCHAR   szMsgBuffer[] = "Hello World from client";

pClient01 = new AGHClient();
status = pClient01->SwCall(szMsgBuffer, &Resp);
if(status == SwOperationSucceed)
{
   /* ok */
   cout << "SwCall response: " << Resp << endl;
}
else
{
   /* failed */
   cout << "SwCall error response: " << Resp << endl;
}
⋮

pClient01->SwXmlBufferFree(Resp);
⋮

delete pClient01
```

AGHClient::SwXmlBufferFree

This method is called by the client to release the response buffer allocated by an SwCall, SwACall, SwAWait, AGHLFTCmd, or AGHFCall method.

Syntax:

```
VOID SwXmlBufferFree(VOID * pBuffer);
```

where:

pBuffer **(input)**
  A pointer to the buffer to be freed.

Coding example:

```
⋮

#include <aghcaswd.hpp>
⋮

AGHClient pClient01=NULL;
AGHCHAR   szMsgBuffer[] = "Hello World from client";
AGHSTRING Resp;
pClient01 = new AGHClient();
⋮

pClient01->SwCall(szMsgBuffer,&Resp);
⋮

pClient01->SwXmlBufferFree(Resp);
delete pClient01;
```

AGHServer

The class AGHServer encapsulates a server object. The server class definitions are in the header file AGHCASVD.

Constructors

When you create your first server object, an internal logger service object is created that writes trace and log entries. If you create a second server object, and if you want both to use the same logging and trace data sets, you must pass the logger service object from the first server object. To get the logger service object, use the getLoggerObj method as shown in the example below.

Syntax:

```
AGHServer(CHAR * pszServerName, AGHLoggerService * pLog = NULL);
AGHServer(AGHLoggerService * pLog = NULL);
```

where:

pszServerName **(input)**
: A pointer to a null-terminated string containing the name of the server object. If you do not specify a name, you can specify one later in one of the following ways:
- Via the setServerName method
- During initialization

If you don't specify a name, the server name is set to SERVER01 during the first request retrieval (see "AGHServer::retrieve" on page 46). Whether you specify a name or use the default name, your profile data set must contain a profile for a server with that name (see "Chapter 3. Creating client and server profiles" on page 21).

pLog **(input)**
: A pointer to the logger object of a previously created AGHServer object. If this is not specified, a new logger object is created for the server.

Coding example:

```
#include <aghcasvd.hpp>
⋮

AGHServer * pServer01;
AGHServer * pServer02;
⋮

pServer01 = new AGHServer("SERVER01");
⋮

pServer02 = new AGHServer("SERVER02", pServer01->getLoggerObj());
⋮
```

Methods

AGHServer::clientHub

Starts a client hub. For a description of a client hub, see "Client hub and server request handler" on page 9.

If a name was already specified for a client hub, either when it was created or via the init method (see "AGHServer" on page 39), do not specify a name with the clientHub method; any name you specify will be ignored. If a name for the client hub was not already specified, specify a name with the clientHub method, otherwise the default name SERVER01 will be used. Whether you specify a name or use the default name, your profile data set must contain a profile for a client hub with that name (see "Chapter 3. Creating client and server profiles" on page 21).

Syntax:

```
LONG clientHub();
LONG clientHub(CHAR * pszServerName);
```

where:

pszServerName
: A pointer to a null-terminated string containing the name of a server profile in the profile data set (see "Chapter 3. Creating client and server profiles" on page 21).

Return values: The condition code AGHOK (0) indicates that the operation was successful; the condition code 2033 indicates that the request queue is empty; any other condition code usually indicates an error (see Table 11 on page 104).

Coding example:

```
#include <aghcasvd.hpp>
⋮

AGHServer pServer01;
LONG      lCCode;
⋮

pServer01 = new AGHServer("HUB01");
⋮

lCCode = pServer01->clientHub();
⋮

delete pServer01;
```

AGHServer::getConditionCode

MQSeries Adapter saves the last condition code that it issues. A server can use this method to retrieve this condition code. Possible values are described in Table 11 on page 104.

Syntax:

```
LONG getConditionCode();
```

Coding example:

```
⋮

#include <aghcasvd.hpp>
⋮

AGHServer * pServer01 = NULL;
LONG        lCCode;
⋮

pServer01 = new AGHServer();
⋮

lCCode = pServer01->getConditionCode();
⋮
```

AGHServer::getCorrespondentName

This method returns a pointer to a null-terminated string containing the correspondent name.

Syntax:

```
CHAR * getCorrespondentName();
```

AGHServer::getErrorMessage

This method returns a pointer to the error message that MQSeries Adapter last issued. The error messages are described in “Appendix A. Messages and codes” on page 95. The maximum size of the error message buffer is 256 bytes.

Syntax:

```
CHAR * getErrorMessage();
```

Coding example:

```
⋮
#include <aghcasvd.hpp>
⋮
AGHServer * pServer01 = NULL;
CHAR *      pszErrorMessage;
⋮
pServer01 = new AGHServer();
⋮
pszErrorMessage = pServer01->getErrorMessage();
⋮
delete pServer01;
```

AGHServer::getLog

This method returns a pointer to a one-character string that indicates whether message logging is set on (Y) or off (N).

Syntax:

```
CHAR * getLog();
```

AGHServer::getLoggerObj

This method returns a pointer to the logger object of the server instance. A single logger object can be shared by several servers.

Syntax:

```
AGHLoggerObject * getLoggerObj();
```

For a coding example, see “Constructors” on page 40.

AGHServer::getMqmName

This method returns a pointer to a null-terminated string containing the MQSeries queue manager name. The MqmName can only be set as a profile parameter.

Syntax:

```
CHAR * getMqmName();
```

AGHServer::getReplyToQueueName

This method returns a pointer to a null-terminated string containing the MQSeries reply-to queue name. The reply-to queue name cannot be set; its name is taken from the information provided with a request message.

Syntax:

```
CHAR * getReplyToQueueName();
```

AGHServer::getRequestQueueName

This method returns a pointer to a null-terminated string containing the MQSeries request queue name. The request queue name can only be set as a profile parameter.

Syntax:

```
CHAR * getRequestQueueName();
```

AGHServer::getServerName

This method returns a pointer to a null-terminated string containing the server name. The server name can be set only before the AGHServer::init method has been executed.

Syntax:

```
CHAR * getServerName();
```

AGHServer::getTraceLevel

Returns the current trace level. See “Tracing” on page 90 for more information.

Syntax:

```
ULONG getTraceLevel();
```

AGHServer::getUserId

This method returns a pointer to a null-terminated string containing the current user ID.

Syntax:

```
CHAR * getUserId();
```

AGHServer::init

This method is used to initialize a server object that was created using the server constructor, but for which no server name parameter was specified. The parameters in the profile for the server being initialized overwrite any parameters that were set by other means, for example using set methods.

The default server name is SERVER01, so if you initialize a server without specifying a name, your profile data set must contain a profile with the name SERVER01.

Syntax:

```
LONG init();
LONG init(CHAR * pszServerName);
```

where:

pszServerName **(input)**
: A pointer to a null-terminated character string containing the name of a server profile in the profile data set (see “Chapter 3. Creating client and server profiles” on page 21).

Return values: The condition code AGHOK (0) indicates that the operation was successful; any other condition code usually indicates an error (see Table 11 on page 104).

Coding example:

```
#include <aghcasvd.hpp>
⋮

AGHServer *pServer01;
LONG       lCCode;
⋮

pServer01 = new AGHServer();
lCCode = pServer01->init("SERVER01");
⋮
```

AGHServer::release

This method releases the message buffer created by the **retrieve** method (see "AGHServer::retrieve" on page 46).

Syntax:

```
LONG release(AGHSTRING req);
```

where:

req **(input)**
: A pointer to the message buffer to be released; this pointer was returned with the **retrieve** method.

Return values: The condition code AGHOK (0) indicates that the operation was successful; any other condition code usually indicates an error (see Table 11 on page 104).

Coding example:

```
#include <aghcasvd.hpp>
⋮

AGHServer * pServer01;
LONG        lCCode;
AGHSTRING   req;
⋮

pServer01 = new AGHServer("SERVER01");
⋮

lCCode=pServer01->retrieve(&req);
⋮

lCCode = pServer01->release(req);
⋮

delete pServer01;
```

AGHServer::reply

This method places a response into the reply-to queue.

Syntax:

```
LONG reply(AGHSTRING resp);
```

where:

resp **(input)**
: A pointer to the response.

Return values: The condition code AGHOK (0) indicates that the operation was successful; any other condition code usually indicates an error (see Table 11 on page 104).

Coding example:

```
#include <aghcasvd.hpp>
⋮

AGHServer * pServer01;
LONG        lCCode;
AGHSTRING   resp;
⋮

resp = new szBuffer[512];
strcpy(szBuffer,"response message");
⋮

pServer01 = new AGHServer("SERVER01");
⋮

lCCode = pServer01->reply(resp);
⋮

delete pServer01;
delete resp;
```

AGHServer::requestHandler

Start a server request handler. For a description of a server request handler, see "Client hub and server request handler" on page 9.

If a name was already specified for a server request handler, either when it was created or via the init method (see "AGHServer" on page 39), do not specify a name with the requestHandler method; any name you specify will be ignored. If a name for the server request handler was not already specified, specify a name with the requestHandler method, otherwise the default name SERVER01 will be used. Whether you specify a name or use the default name, your profile data set must contain a profile for a server request handler with that name.

Syntax:

```
LONG requestHandler();
LONG requestHandler(CHAR * pszServerName);
```

where:

pszServerName **(input)**
: A pointer to a null-terminated character string containing the name of a server profile in the profile data set.

Return values: The condition code AGHOK (0) indicates that the operation was successful; the condition code 2033 indicates that the request queue is empty; any other condition code usually indicates an error (see Table 11 on page 104).

Coding example:

```
#include <aghcasvd.hpp>
⋮

AGHServer * pServer01;
LONG        lCCode;
```

```
⋮
pServer01 = new AGHServer("SERVER01");
⋮
lCCode = pServer01->requestHandler();
⋮
delete pServer01;
```

AGHServer::retrieve

This method retrieves the next message from the request queue.

If the server was not initialized when it was created (see "AGHServer" on page 39) or via the init method (see "AGHServer::init" on page 43), the retrieve method initializes the server and assigns it the name SERVER01. In this case, your profile data set must contain a profile with the name SERVER01.

Syntax:

```
LONG retrieve(AGHSTRING * pReq);
```

where:

pReq **(output)**
: A pointer to the address of the retrieved message. After the data is no longer needed, the buffer must be released with the **release** method.

Return values: The condition code AGHOK (0) indicates that the operation was successful; the condition code 2033 indicates that the request queue is empty; any other condition code usually indicates an error (see Table 11 on page 104).

Coding example:

```
#include <aghcasvd.hpp>
⋮
AGHServer * pServer01;
LONG        lCCode;
AGHSTRING   Req;
⋮
pServer01 = new AGHServer("SERVER01");
⋮
lCCode = pServer01->retrieve(&Req);
⋮
lCCode = pServer01->release(&Req);
⋮
delete pServer01;
```

AGHServer::setCorrespondentName

Sets the correspondent name; a null-terminated string with a maximum length of 24 characters can be specified. The correspondent name is used for authorization checking only.

Syntax:

```
LONG setCorrespondentName(CHAR * pszCorrespondent);
```

where:

pszCorrespondent **(input)**
A pointer to a null-terminated string containing the correspondent name.

Return values: The condition code AGHOK (0) indicates that the operation was successful; any other condition code usually indicates an error (see Table 11 on page 104).

AGHServer::setLog

If message logging is activated (that is, if the Log parameter in the server profile is set to Y), you can use this call to switch message logging off and on.

Syntax:

```
LONG setLog(CHAR * pszLog);
```

where:

pszLog **(input)**
A pointer to a null-terminated string containing the character N (to switch logging off) or Y (to switch logging on).

Return values: The condition code AGHOK (0) indicates that the operation was successful; any other condition code usually indicates an error (see Table 11 on page 104).

AGHServer::setServerName

The server name can be set only before the init method is executed. The default name is SERVER01.

Syntax:

```
LONG setServerName(CHAR * pszServerName);
```

where:

pszServerName **(input)**
A pointer to a null-terminated string containing the server name.

Return values: The condition code AGHOK (0) indicates that the operation was successful; any other condition code usually indicates an error (see Table 11 on page 104).

AGHServer::setTraceLevel

Set the trace level for the server. See “Tracing” on page 90 for more information.

Syntax:

```
LONG setTraceLevel(ULONG ulLevel);
```

where:

ulLevel **(input)**
An integer that corresponds to the trace level.

Return values: The condition code AGHOK (0) indicates that the operation was successful; any other condition code usually indicates an error (see Table 11 on page 104).

AGHServer::setUserId

Use this method to set the user ID for which access authority is to be checked (see “Authorization checking” on page 13). If this method is not used, the user ID of the process user is assumed as a default. This method can only be called when running in CICS or IMS environment.

Servers that process messages created by a CICS or IMS user should set the user ID to that of the creating user. The server must retrieve the user ID that started the CICS or IMS transaction. For a client hub, the user ID is taken from the MQMD (message descriptor) of the message.

Syntax:

```
LONG setUserId(CHAR * pszUid);
```

where:

pszUid **(input)**
A pointer to a null-terminated character string containing the user ID that is to be used for all subsequent authorization checking.

Return values: The condition code AGHOK (0) indicates that the operation was successful; any other condition code usually indicates an error (see Table 11 on page 104).

Coding example:

```
#include <aghcasvd.hpp>
⋮

AGHServer * pServer01;
LONG        lCCode;
⋮

pServer01 = new AGHServer("SERVER01");
⋮

lCCode = pServer01->setUserId("USERID");
⋮

delete pServer01;
```

Chapter 6. MQSeries Adapter functions for C and COBOL

The API provided by MQSeries Adapter contains functions for C and COBOL that let clients and servers running on OS/390 systems send requests, transfer files, and process responses. These functions are contained in the following DLLs:

- **AGHCADLC** contains the functions used by **clients**.
- **AGHCADLS** contains the functions used by **servers**.

The condition codes issued by clients and servers are described in “Condition codes” on page 104. To see the accompanying error message, check the trace or retrieve the error message using the AGHClientGetErrorMessage or AGHServerGetErrorMessage functions. If you require more information, consider setting a higher trace level as described in “Tracing” on page 90.

Table 5. Functions used by clients

Name and description	Page
SwXmlBufferFree—Free client buffer	81
AGHClientGetConditionCode—Get condition code for client	50
AGHClientGetErrorMessage—Get error message for client	51
AGHFCall—Initiate a FileAct transfer	56
SwACall—Initiate an asynchronous InterAct transfer	75
SwCall—Initiate a synchronous InterAct transfer	79
AGHLFTCmd—Issue an LFT command	58
SwAWait—Retrieve a response from an asynchronous call	77
AGHClientSetClientName—Set a client name	52
AGHClientSetUserId—Set a client user ID	54

Table 6. Functions used by servers

Name and description	Page
AGHServerGetConditionCode—Get condition code for server	62
AGHServerGetErrorMessage—Get error message for server	63
AGHServerInit—Initialize a server	64
AGHServerReply—Place a response in the reply-to queue	67
AGHServerRelease—Release the server’s message buffer	65
AGHServerRetrieve—Retrieve a request	70
AGHServerSetUserId—Set a user ID for a server	72
AGHServerClientHub—Start a client hub	61
AGHServerRequestHandler—Start a server request handler	69
AGHServerTerm—Terminate server	74

© Copyright IBM Corp. 2001

AGHClientGetConditionCode—Get condition code for client

MQSeries Adapter saves the last condition code that it issues. A client can use this function to retrieve this condition code. Possible values are described in Table 10 on page 104.

Format for C

```
#include <aghcadef.h>
LONG AGHClientGetConditionCode();
```

Coding examples

Coding example for C

```
⋮
#include <aghcadef.h>
⋮
LONG lCCode;
⋮
lCCode = AGHClientGetConditionCode();
⋮
```

Coding example for COBOL

```
⋮
       data division.
       working-storage section.
       01  CCode                   pic s9(9) binary.
⋮
       procedure division.
⋮
       call 'AGHClientGetConditionCode'
             returning       CCode
⋮
```

AGHClientGetErrorMessage—Get error message for client

Use this function to retrieve a pointer to the error message that MQSeries Adapter last issued. The messages are described in “Appendix A. Messages and codes” on page 95. The maximum size of the error message buffer is 256 bytes.

Format for C

```
#include <aghcadef.h>
CHAR * AGHClientGetErrorMessage();
```

Coding examples

Coding example for C

```
⋮
#include <aghcadef.h>
⋮
CHAR * pszErrorMessage;
⋮
pszErrorMessage = AGHClientGetErrorMessage();
⋮
```

Coding example for COBOL

```
⋮
       data division.
       working-storage section.
       77  msgLength              pic s9(9) binary.
       01  pErrorMsg              pointer.
⋮
       linkage divison.
       errorMsg                   pic x(256).
⋮
       procedure division.
⋮
       call 'AGHClientGetErrorMessage'
             returning       pErrorMsg
       set address of errorMsg to pErrorMsg
⋮
```

AGHClientSetClientName—Set a client name

Use this function to overwrite the label for the profile entry that is used to initialize the MQSeries Adapter client. If you overwrite the default name CLIENT01, you must do so as the first call to MQSeries Adapter.

This function returns a condition code: AGHOK (0) if successful; AGHERRORCLIENT (10) if unsuccessful.

Format for C

```
#include <aghcadef.h>
LONG AGHClientSetClientName(CHAR * pszClientName);
```

where:

pszClientName **(input)**
: A pointer to a null-terminated character string containing the client name.

Coding examples

Coding example for C

```
⋮

#include <aghcadef.h>
⋮

CHAR szClientName[] = "CLIENT02";
LONG lCCode;
⋮

lCCode = AGHClientSetClientName(szClientName);
if(lCCode == AGHOK)
{
    /* ok */
}
else {
  /* failed */
}
⋮
```

Coding example for COBOL

```
⋮

      data division.
       working-storage section.
       01  clientName             pic x(32).
       01  CCode                  pic s9(9) binary.
⋮

       procedure division.
⋮

       move z'CLIENT02' to clientName
       call 'AGHClientSetClientName' using      clientName
                                     returning  CCode
       if CCode = AGHOK then
      *  ok
       else
      * failed
```

```
        end-if
  ⋮
```

AGHClientSetUserId—Set a client user ID

Use this function to set the user ID for which an ESM such as RACF is to check access authority (see “Authorization checking” on page 13). This function is valid only when running in a CICS or IMS environment. Clients that process messages created by a CICS or IMS application should set the client user ID to the process user ID of the corresponding CICS or IMS environment.

This function returns a condition code: AGHOK (0) if successful; AGHERRORSECURITY (16) if unsuccessful.

Format for C

```
#include <aghcadef.h>
LONG AGHClientSetUserId(CHAR * pszUserId);
```

where:

pszUserId **(input)**
: A pointer to a null-terminated character string containing the user ID that is to be used for all subsequent authorization checking.

Coding examples

Coding example for C

```
⋮
#include <aghcadef.h>
⋮
CHAR szUserId[8+1]="USERID";
LONG lCCode;
⋮
lCCode = AGHClientSetUserId(szUserId);
if(lCCode == AGHOK)
{
  /* ok */
}
else {
  /* failed */
}
⋮
```

Coding example for COBOL

```
⋮
       data division.
       working-storage section.
       01  userId                pic x(9).
       01  CCode                 pic s9(9) binary.
⋮
       procedure division.
⋮
       call 'AGHClientSetUserId' using      userId
                                 returning  CCode
       if CCode = AGHOK then
      *  ok
       else
```

```
      * failed
       end-if
 ⋮
```

AGHFCall—Initiate a FileAct transfer

This function initiates a FileAct transfer for a file that is already on an SAG workstation. This file might have been moved to that system by an AGHLFTCmd call, but not necessarily. To make sure that the request is sent to the same instance of SAG that processed the corresponding AGHLFTCmd, the same AGHFileHeader structure that was used by that AGHLFTCmd must be used by this function. For more information about local file transfers, see:

- "AGHLFTCmdParm" on page 26
- "AGHFileHeader" on page 27
- "AGHLFTCmd—Issue an LFT command" on page 58

This function returns SwOperationSucceed or SwOperationFailed.

Format for C

```
#include <aghcadef.h>
SwStatus AGHFCall(AGHSTRING req, AGHPSTRING pResp, AGHFileHeader* fhead);
```

where:

req **(input)**
: The name of the file stored on the SAG workstation, and that is to be transferred via the SIPN to a remote location (that is, for which a FileAct is to be started). For information on how to specify such names, refer to the *SWIFTAlliance Gateway MQHA Application Programming Guide Release 1.2.0*.

pResp **(output)**
: A pointer to an area containing the address of another area where the response is to be stored (the response buffer). MQSeries Adapter allocates memory for the response buffer; the program must free this memory after it is no longer needed.

fhead **(input/output)**
: A pointer to the AGHFileHeader structure, which contains the name of the queue manager and the name of the target queue to which this request must be sent (see "AGHFileHeader" on page 27).

Coding examples

Coding example for C

```
⋮

#include <stdio.h>
#include <aghcadef.h>
⋮

AGHFileHeader  file01;
AGHCHAR        szReqBuffer[512];
AGHSTRING      Response;
SwStatus       status;

memset(szReqBuffer, '\0', sizeof(szReqBuffer));
strcpy(szReqBuffer, "x:/path/file.ext");

status = AGHFCall(szReqBuffer, &Response, &file01);
if(status == SwOperationSucceed)
{
   /* ok */
   printf("AGHFCall response: %s\n", Response);
}
```

```
else {
    /* failed */
   printf("AGHFCall error response: %s\n", Response);
}
⋮

SwXmlBufferFree(Response);
⋮
```

Coding example for COBOL

```
⋮

       data division.
       working-storage section.
       copy agheacb1.
       77  msgLength               pic s9(9) binary.
       01  requestMsg              pic x(100).
       01  responseMsgAddr         pointer.
       01  status                  pic s9(9) binary.

       linkage section.
       01  responseMsg             pic x(100).

       procedure division.

      *----------------------------------------------------------------*
      *      build the request                                         *
      *----------------------------------------------------------------*
           move z'x:/path/file.ext' to requestMsg

           call 'AGHFCall' using   requestMsg
                                   responseMsgAddr
                                   AGHFileHeader
               returning           status

           if status = SwOperationSucceed then
      *        ok
               set address of responseMsg to responseMsgAddr
⋮

           else
      *        failed
⋮

           end-if

      *----------------------------------------------------------------*
      *          release the response's storage                        *
      *----------------------------------------------------------------*
           call 'SwXmlBufferFree' using responseMsgAddr
⋮
```

AGHLFTCmd—Issue an LFT command

This function is used to put files onto an SAG workstation (for example, in preparation for a FileAct transfer), or to get, list, or delete such files.

This function returns SwOperationSucceed or SwOperationFailed.

Format for C

```
#include <aghcadef.h>
SwStatus AGHLFTCmd(AGHSTRING req, AGHPSTRING pResp, AGHFileHeader* fhead);
```

where:

req **(input)**
: A pointer to the local file transfer (LFT) command to be issued. These commands are described in "AGHLFTCmdParm" on page 26.

pResp **(output)**
: A pointer to an area containing the address of another area where the response is to be stored (the response buffer). The contents of this buffer can be read using the AGHLFTCmdParm data structure (see "AGHLFTCmdParm" on page 26). MQSeries Adapter allocates memory for the response buffer; the program must free this memory after it is no longer needed.

fhead **(input/output)**
: The address of the data structure containing the length and address of the data to be put (see "AGHFileHeader" on page 27). If a PUT command is issued in preparation for a FileAct request (AGHFCall), information needed by the FileAct request (for example, the names of the queue manager and the request queue to be used) is returned in this data structure.

Coding examples

Coding example for C

```
⋮

#include <stdio.h>
#include <aghcadef.h>
⋮

AGHFileHeader  file01;
AGHLFTCmdParm  *pParm01;
AGHCHAR        szReqBuffer[512];
AGHSTRING      Response=NULL;
CHAR szFileBuffer[]="data to be moved to file on SAG workstation";
SwStatus       status;

file01.lDataLength = sizeof(szFileBuffer);
file01.pData       = szFileBuffer;
pParm01 = (AGHLFTCmdParm*)szReqBuffer;

memset(szReqBuffer, ' ', sizeof(szReqBuffer));
memcpy(pParm01->cLFTVersion, AGHLFTVERSION,
                             sizeof(pParm01->cLFTVersion));
memcpy(pParm01->cLFTCmd, "PUT", 3);
strcpy(pParm01->cLFTText, "x:/path/file.ext");

status = AGHLFTCmd(szReqBuffer, &Response, &file01);
```

```
if(status == SwOperationSucceed)
    /* ok */
   printf("AGHLFTCmd response: %s\n", Response);
}
else {
    /* failed */
   printf("AGHLFTCmd error response: %s\n", Response);
}

SwXmlBufferFree(Response);
⋮
```

Coding example for COBOL

```
⋮

       data division.
       working-storage section.
       copy agheacb1.
       77  msgLength              pic s9(9) binary.
       01  requestMsg             pic x(100).
       01  fileData               pic x(100).
       01  responseMsgAddr        pointer.
       01  status                 pic s9(9) binary.
       01  pErrorMsg              pointer.
       01  CCode                  pic s9(9) binary.

       linkage section.
       01  responseMsg            pic x(100).
       01  errorMsg               pic x(256).

       procedure division.

      *-----------------------------------------------------------------*
      *      build the request                                          *
      *-----------------------------------------------------------------*
           move z'data to be moved to file on SAG workstation' to fileData
           move z'' to requestMsg

           set pData of AGHFileHeader to address of fileData
           inspect fileData tallying msgLength
                                  for characters before x'00'
           move msgLength to lDataLength of AGHFileHeader

           move AGHLFTVERSION to cLFTVersion    of AGHLFTCmdParm
           move z'PUT    '    to cLFTCmd        of AGHLFTCmdParm
           move z'x:/path/file.ext' to cLFTText of AGHLFTCmdParm

           call 'AGHLFTCmd' using  requestMsg
                                  responseMsgAddr
                                  AGHFileHeader
               returning          status

           if status = SwOperationSucceed then
      *-----------------------------------------------------------------*
      *          handle the response                                    *
      *-----------------------------------------------------------------*
               set address of responseMsg to Resp of AGHResponse
               move 0 to          msgLength
               inspect responseMsg tallying msgLength
                                  for characters before x'00'
               display '..response has length ' msglength ':'
               display responseMsg(1:msgLength)
               move 0 to return-code
           else
               display 'AGHLFTCmd returned ' status
               call 'AGHClientGetConditionCode'
```

```
                returning       CCode
            call 'AGHClientGetErrorMessage'
                returning       pErrorMsg
            set address of errorMsg to pErrorMsg
            move 0 to           msgLength
            inspect errorMsg tallying msgLength
                                for characters before x'00'
            display 'AGHClient condition code=' CCode
            display 'AGHClient error message='  errorMsg(1:msgLength)
            move 8 to return-code
        end-if
      *----------------------------------------------------------------*
      *          release the response's storage                        *
      *----------------------------------------------------------------*
        call 'SwXmlBufferFree' using responseMsgAddr
  :
```

AGHServerClientHub—Start a client hub

For a description of a client hub, see "Client hub and server request handler" on page 9.

For a client hub, the user ID used for authorization checking is taken from the MQMD (message descriptor) of the message.

This function returns a condition code. The condition code AGHOK (0) indicates that the operation was successful. The condition code 2033 indicates that the request queue is empty. Other condition codes usually indicate an error (see "Condition codes" on page 104).

Format for C

```
#include <aghcadef.h>
LONG  AGHServerClientHub(CHAR * pszServerName);
```

where:

pszServerName
: A pointer to a null-terminated string containing the name of a server profile in the profile data set (see "Chapter 3. Creating client and server profiles" on page 21).

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮

LONG      lCCode;
⋮

lCCode = AGHServerClientHub("HUB01");
⋮
```

Coding example for COBOL

```
⋮

     data division.
     working-storage section.
⋮

       01  szServerName           pic x(32).
       01  CCode                  pic s9(9) binary.
⋮

     procedure division.

           move z'HUB01' to szServerName
           call 'AGHServerClientHub' using      szServerName
                                     returning  CCode
⋮
```

AGHServerGetConditionCode—Get condition code for server

MQSeries Adapter saves the last condition code that it issues. A server can use this function to retrieve and return this condition code. Possible values are described in "Condition codes" on page 104.

Format for C

```
#include <stdio.h>
#include <aghcadef.h>
LONG  AGHServerGetConditionCode(AGHSRVHANDLE handle);
```

where:

handle **(input)**
A pointer to the handle returned with the AGHServerInit call.

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮

AGHSRVHANDLE pServer01;
LONG      lCCode;
⋮

pServer01 = AGHServerInit("SERVER01");
⋮

lCCode = AGHServerGetConditionCode(pServer01);
printf("Error Code: %d\n",lCCode);
⋮
```

Coding example for COBOL

```
⋮

       data division.
       working-storage section.
⋮

         01  szServerName           pic x(32).
         01  pServer                pointer.
         01  CCode                  pic s9(9) binary.
⋮

       procedure division.

             move z'SERVER01' to szServerName
             call 'AGHServerInit' using      szServerName
                                  returning  pServer
⋮

             call 'AGHServerGetConditionCode' using      pServer
                                              returning  CCode
⋮
```

AGHServerGetErrorMessage—Get error message for server

Use this function to retrieve a pointer to the error message that MQSeries Adapter last issued. The messages are described in “Appendix A. Messages and codes” on page 95. The maximum size of the error message buffer is 256 bytes.

Format for C

```
#include <aghcadef.h>
CHAR * AGHServerGetErrorMessage(AGHSRVHANDLE handle);
```

where:

handle **(input)**
: A pointer to the handle returned with the AGHServerInit call.

Coding examples

Coding example for C

```
#include <stdio.h>
#include <aghcadef.h>
⋮

AGHSRVHANDLE pServer01;
CHAR *       pszErrMsg;
⋮

pServer01 = AGHServerInit("SERVER01");
⋮

pszErrMsg = AGHServerGetErrorMessage(pServer01);
printf("Error: %s\n",pszErrMsg);
⋮
```

Coding example for COBOL

```
⋮

     data division.
     working-storage section.
⋮

       01  szServerName           pic x(32).
       01  pServer                pointer.
       01  pErrMsg                pointer.
⋮

     procedure division.

           move z'SERVER01' to szServerName
           call 'AGHServerInit' using      szServerName
                                returning  pServer
⋮

           call 'AGHServerGetErrorMessage' using      pServer
                                           returning  pErrMsg
⋮
```

AGHServerInit—Initialize a server

This function initializes a server and must be the first function that is called. Your profile data set must contain a profile with the name specified. This function returns a handle that is used as an input parameter for other functions.

Format for C

```
#include <aghcadef.h>
AGHSRVHANDLE AGHServerInit(CHAR * pszServerName);
```

where:

pszServerName **(input)**
: A pointer to a null-terminated string containing the name of a server profile in the profile data set (see "Chapter 3. Creating client and server profiles" on page 21).

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮

CHAR         szServerName[32];
AGHSRVHANDLE pServer01;
⋮

strcpy(szServerName,"SERVER01");
pServer01 = AGHServerInit(szServerName);
⋮
```

Coding example for COBOL

```
⋮

     data division.
     working-storage section.
⋮

        01  szServerName            pic x(32).
        01  pServer                 pointer.
⋮

     procedure division.

            move z'SERVER01' to szServerName
            call 'AGHServerInit' using      szServerName
                                 returning  pServer
⋮
```

AGHServerRelease—Release the server's message buffer

This function releases the message buffer created for the AGHServerRetrieve call (see "AGHServerRetrieve—Retrieve a request" on page 70).

The condition code AGHOK (0) indicates that the operation was successful (that is, that the message buffer was released).

Format for C

```
#include <aghcadef.h>
LONG  AGHServerRelease(AGHSRVHANDLE handle, AGHSTRING req);
```

where:

handle **(input)**
A pointer to the handle returned with the AGHServerInit call.

req **(input)**
A pointer to the message buffer to be released; this pointer was returned with the AGHServerRetrieve call.

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮

AGHSRVHANDLE pServer01;
AGHSTRING Req;
LONG      lCCode;
⋮

pServer01 = AGHServerInit("SERVER01");
lCCode = AGHServerRetrieve(pServer01,&Req);
⋮

lCCode = AGHServerRelease(pServer01,Req);
⋮
```

Coding example for COBOL

```
⋮

     data division.
     working-storage section.
⋮

           01  szServerName            pic x(32).
           01  pServer                 pointer.
           01  requestMsgAddr          pointer.
           01  CCode                   pic s9(9) binary.

           linkage section.
           01  requestMsg              pic x(100).
⋮

           procedure division.

           move z'SERVER01' to szServerName
           call 'AGHServerInit' using     szServerName
                                returning pServer
```

```
⋮
    call 'AGHServerRetrieve' using     pServer
                                       requestMsgAddr
                             returning CCode
        set address of requestMsg to requestMsgAddr
⋮
    call 'AGHServerRelease' using     pServer
                                      requestMsg
                            returning CCode
```

AGHServerReply—Place a response in the reply-to queue

This function places a response into the reply-to queue.

The condition code AGHOK (0) indicates that the operation was successful, and that a response was put into the reply-to queue. Any other condition code indicates an error (see “Condition codes” on page 104).

Format for C

```
#include <aghcadef.h>
LONG  AGHServerReply(AGHSRVHANDLE handle, AGHSTRING Resp);
```

where:

handle **(input)**
: A pointer to the handle returned with the AGHServerInit call.

Resp **(input)**
: A pointer to the response.

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮

AGHSRVHANDLE pServer01;
LONG       lCCode;
⋮

pServer01 = AGHServerInit("SERVER01");
⋮

lCCode = AGHServerReply(pServer01,"Hello client");
⋮
```

Coding example for COBOL

```
⋮

     data division.
     working-storage section.
⋮

       01  szServerName           pic x(32).
       01  pServer                pointer.
       01  responseMsg            pic x(100).
       01  CCode                  pic s9(9) binary.
⋮

     procedure division.

           move z'SERVER01' to szServerName
           call 'AGHServerInit' using     szServerName
                                returning pServer
⋮

           move z'Hello client' to responseMsg
           call 'AGHServerReply' using      pServer
                                            responseMsg
```

```
                    returning CCode
    ⋮
```

AGHServerRequestHandler—Start a server request handler

For a description of a server request handler, see "Client hub and server request handler" on page 9.

This function returns a condition code. The condition code AGHOK (0) indicates that the operation was successful. The condition code 2033 indicates that the request queue is empty. Other condition codes usually indicate an error (see "Condition codes" on page 104).

Format for C

```
#include <aghcadef.h>
LONG  AGHServerRequestHandler(CHAR * pszServerName);
```

where:

pszServerName **(input)**
: A pointer to a null-terminated string containing the name of a server profile in the profile data set.

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮
LONG     lCCode;
⋮
lCCode = AGHServerRequestHandler("SERVER01");
⋮
```

Coding example for COBOL

```
⋮
     data division.
     working-storage section.
⋮
       01  szServerName           pic x(32).
       01  CCode                  pic s9(9) binary.
⋮
       procedure division.

           move z'SERVER01' to szServerName
⋮
           call 'AGHServerRequestHandler' using      szServerName
                                          returning  CCode
⋮
```

AGHServerRetrieve—Retrieve a request

This function retrieves the next message from the request queue.

This function returns a condition code. A condition code of 0 indicates successful operation. The condition code 2033 indicates that the request queue is empty and there are no further messages to be processed. Other condition codes usually indicate an error (see “Condition codes” on page 104).

Format for C

```
#include <aghcadef.h>
LONG  AGHServerRetrieve(AGHSRVHANDLE handle, AGHSTRING * pReq);
```

where:

handle **(input)**
: A pointer to the handle returned with the AGHServerInit call.

pReq **(output)**
: A pointer to the address of the retrieved message. After the data is no longer needed, the buffer must be released with the AGHServerRelease function.

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮

AGHSRVHANDLE pServer01;
AGHSTRING    Req;
LONG         lCCode;
⋮

pServer01 = AGHServerInit("SERVER01");
⋮

lCCode = AGHServerRetrieve(pServer01,&Req);
⋮
```

Coding example for COBOL

```
⋮

      data division.
      working-storage section.
⋮

      01  szServerName           pic x(32).
      01  pServer                pointer.
      01  requestMsgAddr         pointer.
      01  CCode                  pic s9(9) binary.

      linkage section.
      01  requestMsg             pic x(100).
⋮

      procedure division.

      move z'SERVER01' to szServerName
      call 'AGHServerInit' using      szServerName
```

```
                            returning  pSserver
⋮

     call 'AGHServerRetrieve' using      pServer
                                         requestMsgAddr
                          returning  CCode
         set address of requestMsg to requestMsgAddr
```

AGHServerSetUserId—Set a user ID for a server

This function sets the user ID for which authorization to access protected resources is checked. This function can only be called when running in CICS or IMS environment. Applications processing messages created by CICS or IMS users should set the user ID of the creating user. The application must retrieve the user ID which has started the CICS or IMS transaction.

This function returns a condition code. The condition code AGHOK (0) indicates that the operation was successful; other condition codes usually indicate an error (see “Condition codes” on page 104).

Format for C

```
#include <aghcadef.h>
LONG AGHServerSetUserId(AGHSRVHANDLE handle, CHAR * pszUserId);
```

where:

handle **(input)**
: A pointer to the handle returned with the AGHServerInit call.

pszUserId **(input)**
: A pointer to a null-terminated string containing the user ID that is to be used for all subsequent authorization checking.

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮

AGHSRVHANDLE pServer01;
LONG         lCCode;
⋮

pServer01 = AGHServerInit("SERVER01");
⋮

lCCode = AGHServerSetUserId(pServer01,"USERID");
⋮
```

Coding example for COBOL

```
⋮

     data division.
     working-storage section.
⋮

       01  szServerName           pic x(32).
       01  pServer                pointer.
       01  szUserId               pic x(9).
       01  CCode                  pic s9(9) binary.
⋮

     procedure division.

           move z'SERVER01' to szServerName
           call 'AGHServerInit' using     szServerName
                                returning pServer
```

```
⋮
          move z'USERID' to szUserId
          call 'AGHServerSetUserId' using      pServer
                                               szUserId
                                    returning  CCode
⋮
```

AGHServerTerm—Terminate server

This function terminates a server and frees all resources used by the server. It must be the last call. It returns a condition code that indicates whether the operation was successful (see “Condition codes” on page 104).

Format for C

```
#include <aghcadef.h>
LONG  AGHServerTerm(AGHSRVHANDLE handle);
```

where:

`handle` **(input)**
: A pointer to the handle returned with the AGHServerInit call.

Coding examples

Coding example for C

```
#include <aghcadef.h>
⋮

AGHSRVHANDLE pServer01;
LONG      lCCode;
⋮

pServer01 = AGHServerInit("SERVER01");
⋮

lCCode = AGHServerTerm(pServer01);
⋮
```

Coding example for COBOL

```
⋮

     data division.
     working-storage section.
⋮

       01  szServerName           pic x(32).
       01  pServer                pointer.
       01  CCode                  pic s9(9) binary.
⋮

     procedure division.
           move z'SERVER01' to szServerName
           call 'AGHServerInit' using      szServerName
                                returning  pServer
⋮

           call 'AGHServerTerm' using      pServer
                                returning  CCode
⋮
```

SwACall—Initiate an asynchronous InterAct transfer

This function is used to send a request that is to be treated asynchronously; that is, control is to be passed back to the client immediately after the request has been added to the request queue. To retrieve the response associated with a request submitted using this call, use the SwAWait function (see "SwAWait—Retrieve a response from an asynchronous call" on page 77).

This function returns SwOperationSucceed or SwOperationFailed.

Format for C

```
#include <aghcadef.h>
SwStatus SwACall(AGHSTRING req, AGHResponse* pResp);
```

where:

req **(input)**
: The address of the area containing the message to be sent (the request buffer). This message can be a basic or an SNL message.

pResp **(output)**
: The address of the AGHResponse structure where the message ID and code page information from SwACall is stored (see "Data structures" on page 25). In case of an error, this structure contains a pointer to the error buffer. MQSeries Adapter allocates memory for the error buffer; the program must free this memory after it is no longer needed.

Coding examples

Coding example for C

```
⋮

#include <stdio.h>
#include <aghcadef.h>

AGHResponse AsyncResp;
SwStatus    status;
AGHCHAR     szMsgBuffer[] = "Hello World from client";

status = SwACall(szMsgBuffer, &AsyncResp);
if(status == SwOperationSucceed)
{
   /* ok */
}
else
{
   /* failed */
   printf("SwACall error response: %s\n", AsyncResp.Resp);
}
SwXmlBufferFree(AsyncResp.Resp);
⋮
```

Coding example for COBOL

```
⋮

       data division.
       working-storage section.
       copy agheacb1.
       77  msgLength               pic s9(9) binary.
       01  requestMsg              pic x(100).
```

```
 01  responseMsgAddr        pointer.
 01  status                 pic s9(9) binary.
 01  pErrorMsg              pointer.
 01  CCode                  pic s9(9) binary.

 linkage section.
 01  responseMsg            pic x(100).
 01  errorMsg               pic x(256).

 procedure division.

*----------------------------------------------------------------*
*      build the request                                         *
*----------------------------------------------------------------*
     move z'Hello world from client.' to requestMsg

     call 'SwACall' using      requestMsg
                               AGHResponse
                  returning  status

     if status = SwOperationSucceed then
*----------------------------------------------------------------*
*          handle the response                                    *
*----------------------------------------------------------------*
         set address of responseMsg to Resp of AGHResponse
         move 0 to           msgLength
         inspect responseMsg tallying msgLength
                             for characters before x'00'
         display '..response has length ' msglength ':'
         display responseMsg(1:msgLength)
         move 0 to return-code
     else
         display 'SwACall returned ' status
         call 'AGHClientGetConditionCode'
               returning       CCode
         call 'AGHClientGetErrorMessage'
               returning       pErrorMsg
         set address of errorMsg to pErrorMsg
         move 0 to           msgLength
         inspect errorMsg tallying msgLength
                             for characters before x'00'
         display 'AGHClient condition code=' CCode
         display 'AGHClient error message='  errorMsg(1:msgLength)
         move 8 to return-code
     end-if

*----------------------------------------------------------------*
*          release the response's storage                        *
*----------------------------------------------------------------*
     call 'SwXmlBufferFree' using Resp of AGHResponse
```
⋮

SwAWait—Retrieve a response from an asynchronous call

This function retrieves the response that resulted from a previously issued SwACall. This function uses as a handle the pointer to the area containing the address of the response area that was returned by SwACall.

This function returns SwOperationSucceed or SwOperationFailed.

Format for C

```
#include <aghcadef.h>
SwStatus SwAWait(AGHResponse * pResp);
```

where:

pResp **(input/output)**
: The address of the AGHResponse structure that contains a pointer to the response from SwAWait or, in case of an error, the address of the error buffer. MQSeries Adapter allocates memory for the response or error buffer; the program must free this memory after it is no longer needed.

Coding examples

Coding example for C

```
⋮

#include <stdio.h>
#include <aghcadef.h>

AGHResponse AsyncResp;
SwStatus    status;
AGHCHAR     szMsgBuffer[] = "Hello World from client";
⋮

status = SwACall(szMsgBuffer, &AsyncResp);
⋮

status = SwAWait(&AsyncResp);
if(status == SwOperationSucceed)
{
   /* ok */
   printf("SwAWait response: %s\n", AsyncResp.Resp);
}
else
{
   /* failed */
   printf("SwAWait error response: %s\n", AsyncResp.Resp);
}
⋮

SwXmlBufferFree(AsyncResp.Resp);
⋮
```

Coding example for COBOL

```
⋮

       data division.
       working-storage section.
       copy agheacb1.
       77  msgLength              pic s9(9) binary.
       01  status                 pic s9(9) binary.
       01  pErrorMsg              pointer.
```

```
 01  CCode                  pic s9(9) binary.

 linkage section.
 01  responseMsg            pic x(100).
 01  errorMsg               pic x(256).

 procedure division.
```

⋮

```
     call 'SwAWait' using      AGHResponse
                    returning  status

     if status = SwOperationSucceed then
*----------------------------------------------------------------*
*         handle the response                                     *
*----------------------------------------------------------------*
         set address of responseMsg to Resp of AGHResponse
         move 0 to          msgLength
         inspect responseMsg tallying msgLength
                            for characters before x'00'
         display '..response has length ' msglength ':'
         display responseMsg(1:msgLength)
         move 0 to return-code
     else
         display 'SwAWait returned ' status
         call 'AGHClientGetConditionCode'
               returning       CCode
         call 'AGHClientGetErrorMessage'
               returning       pErrorMsg
         set address of errorMsg to pErrorMsg
         move 0 to          msgLength
         inspect errorMsg tallying msgLength
                            for characters before x'00'
         display 'AGHClient condition code=' CCode
         display 'AGHClient error message='  errorMsg(1:msgLength)
         move 8 to return-code
     end-if

*----------------------------------------------------------------*
*         release the response's storage                         *
*----------------------------------------------------------------*
     call 'SwXmlBufferFree' using Resp of AGHResponse
```

⋮

SwCall—Initiate a synchronous InterAct transfer

This function is used to pass a request to MQSeries Adapter that is to be treated synchronously; that is, control is not to be passed back to the client until after SAG returns the response. It is compatible with the native SAG interface.

This function returns SwOperationSucceed or SwOperationFailed.

Format for C

```
#include <aghcadef.h>
SwStatus SwCall(AGHSTRING req, AGHPSTRING pResp);
```

where:

req **(input)**
: The address of the area containing the message to be sent (the request buffer). This message can be a basic or an SNL message.

pResp **(input)**
: A pointer to the address of the response buffer. MQSeries Adapter allocates memory for the response buffer; the program must free this memory after it is no longer needed.

Coding examples

Coding example for C

```
⋮

#include <aghcadef.h>

AGHSTRING Resp;
SwStatus  status;
AGHCHAR   szMsgBuffer[] = "Hello World from client";

status = SwCall(szMsgBuffer, &Resp);
if(status == SwOperationSucceed)
{
   /* ok */
   printf("SwCall  response: %s\n", Resp);
}
else
{
   /* failed */
   printf("SwCall error response: %s\n", Resp);
}
SwXmlBufferFree(Resp);
⋮
```

Coding example for COBOL

```
⋮

       data division.
       working-storage section.
       copy agheacbl.
       77  msgLength              pic s9(9) binary.
       01  requestMsg             pic x(100).
       01  responseMsgAddr        pointer.
       01  status                 pic s9(9) binary.
       01  pErrorMsg              pointer.
       01  CCode                  pic s9(9) binary.
```

```
 linkage section.
 01  responseMsg             pic x(100).
 01  errorMsg                pic x(256).

 procedure division.

*----------------------------------------------------------------*
*      build the request                                         *
*----------------------------------------------------------------*
     move z'Hello world from client.' to requestMsg

     call 'SwCall' using      requestMsg
                              responseMsgAddr
                  returning  status

     if status = SwOperationSucceed then
*----------------------------------------------------------------*
*          handle the response                                    *
*----------------------------------------------------------------*
         set address of responseMsg to responseMsgAddr
         move 0 to           msgLength
         inspect responseMsg tallying msgLength
                             for characters before x'00'
         display '..response has length ' msglength ':'
         display responseMsg(1:msgLength)
         move 0 to return-code
     else
         display 'SwCall returned ' status
         call 'AGHClientGetConditionCode'
               returning       CCode
         call 'AGHClientGetErrorMessage'
               returning       pErrorMsg
         set address of errorMsg to pErrorMsg
         move 0 to           msgLength
         inspect errorMsg tallying msgLength
                             for characters before x'00'
         display 'AGHClient condition code=' CCode
         display 'AGHClient error message='  errorMsg(1:msgLength)
         move 8 to return-code
     end-if

*----------------------------------------------------------------*
*          release the response's storage                        *
*----------------------------------------------------------------*
     call 'SwXmlBufferFree' using responseMsgAddr
```

⋮

SwXmlBufferFree—Free client buffer

This function is called by the client to release the response buffer allocated by an SwCall, SwACall, SwAWait, AGHLFTCmd, or AGHFCall function.

Format for C

```
#include <aghcadef.h>
VOID     SwXmlBufferFree(VOID * pbuffer);
```

where:

pbuffer **(input)**
A pointer to the buffer to be freed.

Coding examples

Coding example for C

```
⋮
#include <aghcadef.h>
⋮
AGHCHAR        szReqBuffer[512];
AGHSTRING      Response;
⋮
SwCall(szReqBuffer,&Response);
⋮
SwXmlBufferFree(Response);
⋮
```

Coding example for COBOL

```
⋮
       data division.
       working-storage section.
⋮
       01  responseMsgAddr         pointer.

       procedure division.
⋮
           call 'SwXmlBufferFree' using responseMsgAddr
⋮
```

Chapter 7. User-written functions

If you elect to use a server request handler (see “Client hub and server request handler” on page 9), you need to provide both an SwCallback function to process the retrieved requests, and an AppXmlBufferFree function to release the response buffer that the SwCallback function allocates.

SwCallback

After the server request handler retrieves a message from the request queue, it calls the user-written function with the name SwCallback, which must be located in the DLL AGHCASCB. This function processes the request and creates a response before returning control to the server request handler. When the server request handler calls SwCallback, it passes to it the following parameters:

- A pointer to the area that contains the request data
- A pointer to the area in which SwCallback is to place the address of the response

Coding examples

The following sections show examples of SwCallback functions.

Coding example for C

⋮

```
#include <stdlib.h>
#include <string.h>
#include <aghcadef.h>
#pragma export (SwCallback)

SwStatus SwCallback( AGHSTRING   RequestToServer,
                     AGHSTRING * pResponseFromServer )
{
 int       intMsgLength;
 SwStatus  status;
 AGHCHAR * pszResponse;

 intMsgLength = strlen(RequestToServer);
 pszResponse = malloc(intMsgLength + 100);
 /* The previously obtained storage must be freed using  */
 /* the function AppXmlBufferFree                        */
 if (pszResponse == NULL)
  {
    status=SwOperationFailed;
  } else
  {
    strcpy(pszResponse, "Server aghcaab1 received ");
    strcat(pszResponse, RequestToServer);
    *pResponseFromServer = pszResponse;
    status=SwOperationSucceed;
  }
 return status;
```

© Copyright IBM Corp. 2001

```
} /* end of SwCallback() */
⋮
```

Coding example for COBOL

⋮

```
       identification division.
       program-id. 'SwCallback'.
       environment division.

       data division.
       working-storage section.
       77  heapid                 pic s9(9) binary.
       77  stgSize                pic s9(9) binary.
       77  msgLength              pic s9(9) binary.
       77  stgAddress             pointer.
       01  feedbackCode.
           03 conditionToken      pic x(8).
           copy ceeigzct.
           03 isInfo              pic s9(9) binary.

       linkage section.
       01  requestMsg             pic x(100).
       01  responseMsgAddr        pointer.
       01  status                 pic s9(9) binary.

       01  responseMsg            pic x(100).

       procedure division using   by reference requestMsg
                                  by reference responseMsgAddr
                                  returning status.

      *----------------------------------------------------------------*
      *     get the request                                            *
      *----------------------------------------------------------------*
           move 0 to              msgLength
           inspect requestMsg     tallying msgLength
                                  for characters before x'00'
      **   display '..requestMsg has length ' msgLength ':'
      **   display requestMsg(1:msgLength)

      *----------------------------------------------------------------*
      *     get storage for the response                               *
      *----------------------------------------------------------------*
           add msgLength 100 giving stgSize
           move 0 to              heapid
           call 'CEEGTST' using   heapid,
                                  stgSize,
                                  stgAddress,
                                  feedbackCode
           if cee000 of feedbackCode then
               set responseMsgAddr to stgAddress
               set address of responseMsg to stgAddress
      *----------------------------------------------------------------*
      *         build the reponse                                      *
      *----------------------------------------------------------------*
               string
                   'Server received '
                   requestMsg(1:msgLength)
                   x'00'
                   delimited by size
                   into           responseMsg
               end-string
               move SwOperationSucceed to status
           else
      **       display 'CEEGTST failed'
```

```
                move SwOperationFailed to status
            end-if

            goback
⋮

        end program 'SwCallback'.
⋮
```

AppXmlBufferFree—Free server buffer

This user-written function is called by the server to release the response buffer allocated by an SwCallback function. This function must be exported in the same DLL as the SwCallback function.

Format

```
#include <aghcadef.h>
VOID     AppXmlBufferFree(VOID * pBuffer);
```

Variables

pBuffer **(input)**
The pointer to the buffer that is to be freed.

Coding examples

The following sections show examples of functions that free a buffer.

Coding example for C

```
⋮

#include <stdlib.h>
#include <aghcctyp.h>
#pragma export (AppXmlBufferFree)
VOID     AppXmlBufferFree(VOID * pBuffer);
⋮

VOID AppXmlBufferFree(VOID * pBuffer)
{
  if (pBuffer <> NULL)
  {
    free(pBuffer);
  }
  return;
}
⋮
```

Coding example for COBOL

```
⋮

        identification division.
        program-id. 'AppXmlBufferFree'.
        environment division.

        data division.
        working-storage section.
        77  stgAddress              pointer.
        01  feedbackCode.
            03 conditionToken       pic x(8).
            copy ceeigzct.
            03 isInfo               pic s9(9) binary.
```

```
 linkage section.
 01  responseMsg             pic x.


 procedure division using    by reference responseMsg.

*-----------------------------------------------------------------*
*      free storage of the response                               *
*-----------------------------------------------------------------*
     set stgAddress to address of responseMsg
     call 'CEEFRST' using    stgAddress,
                             feedbackCode
     if cee000 of feedbackCode then
         goback
     else
**       display 'CEEFRST failed'
         goback
     end-if
⋮

 end program 'AppXmlBufferFree'.
⋮
```

Chapter 8. Message logging and tracing

You can record information while MQSeries Adapter runs by:
- Logging messages
- Setting traces to record error messages and debugging information

Message logging

When message logging is active, MQSeries Adapter copies the contents of each request, plus a log header, to the log data set, system logger stream, or both, depending on which parameters were specified in the MQSeries Adapter profile. The layout of a log record is shown in Table 7. For C and C++ programs, a structure for log records is provided in the include file AGHCG001.HPP.

To activate message logging, set the profile parameter Log=Y (see “Chapter 3. Creating client and server profiles” on page 21), and specify either or both of the following parameters, to indicate to where the log records are to be written:
- LogDDName
- LogStreamName

On the client side, MQSeries Adapter logs a request immediately before passing it to SAG, and logs a response immediately after retrieving it from the reply-to queue. On the server side, MQSeries Adapter logs a request immediately after retrieving it from the request queue, and logs a response immediately before putting it into the reply-to queue.

For an LFT command, the request and the response are logged, but not the contents of the file.

Table 7. Layout of a message log entry

Name	Length (Bytes)	Description
LogHeadLength	4	Length of the log header (96 or X'60')
LogDataLength	4	Length of the log data without the header. The maximum length of each entry is 32000 bytes including the header.
LogRecVersion	4	AGH0
LogRecType	4	Type of log record: **256** Client log record for a request **257** Client log record for a response **320** Server log record for a request **321** Server log record for a response
LogDate	10	Date in the form **YYYY/MM/DD** where: **YYYY** Year **MM** Month (01 to 12) **DD** Date (01 to 31)
(none)	1	Blank between time and date to improve legibility

© Copyright IBM Corp. 2001

Table 7. Layout of a message log entry (continued)

Name	Length (Bytes)	Description
LogTime	12	Time in the form **hh:mm:ss.iii** where: **hh** Hour (00 to 23) **mm** Minute (00 to 59) **ss** Second (00 to 59) **iii** Milliseconds (000 to 999)
(alignment byte)	1	
LogSequence	4	Sequence of log record (used only if the record is too long to fit in one entry)
LogNbOfSeq	4	Number of sequences for the log record (used only if the record is too long to fit in one entry)
LogRecId	4	Log record identifier
LogFinInst	12	Financial institution as set in the profile
LogFinBrch	12	Branch of financial institution as set in the profile
LogFinDept	12	Department of financial institution as set in the profile
LogUserId	8	Client user ID
LogData	LogData Length	Log data

Administering data in the system logger stream

The program AGHCGUTL runs as a batch job and provides functions to process the data from the system logger stream. Its parameter syntax is:

```
PARM=('x logstream "yyyy/mm/dd hh:mm:ss.iii"')
```

where:

x
: A number that indicates the action to be carried out:
  - **1** List
  - **6** Delete
  - **10** Archive
  - **11** Archive and delete

logstream
: The name of the system logger stream that was specified for the LogStreamName parameter in the MQSeries Adapter profile (see “Chapter 3. Creating client and server profiles” on page 21).

yyyy/mm/dd hh:mm:ss.iii
: An optional specification of a year, month (01 to 12), day (01 to 31), hour (00 to 23), minute (00 to 59), second (00 to 59), and milliseconds (000 to 999), used to limit to which records the action applies. Note that you must specify a blank between the date and time. Be sure you specify the date and time correctly, because they are not checked for syntactic correctness, but are instead treated as a simple string.

The following shows examples of how to set the parameters for AGHCGUTL:

PARM=('1 AGH.SWNA1')
: List all records from the system logger stream AGH.SWNA1. The records are printed to SYSOUT.

PARM=('1 AGH.SWNA1 "2001/04/30 12:10:54.000"')
List all records from the system logger stream AGH.SWNA1 that have date and times after the date and time specified. The records are printed to SYSOUT.

PARM=('6 AGH.SWNA1')
Delete all records from the system logger stream AGH.SWNA1.

PARM=('6 AGH.SWNA1 "2001/04/19 13:56:40.000"')
Delete all records from the system logger stream AGH.SWNA1 that have date and times before the date and time specified.

PARM=('10 AGH.SWNA1 "2001/04/30 15:04:35.000"')
Archive records from the system logger stream AGH.SWNA1 that have date and times before the date and time specified. The records are written to a sequential data set allocated to the DD name AGHEGLOG.

PARM=('11 AGH.SWNA1 "2001/04/30 15:04:44.000"')
Archive records from the system logger stream AGH.SWNA1 that have date and times before the date and time specified, then delete the records from the system logger stream. The records are written to a sequential data set allocated to the DD name AGHEGLOG.

Figure 22 shows an example of a job that uses AGHCGUTL to list or delete log records.

```
//*------------------------------------------------------------------
//* Utility to list or delete system logger stream data
//*------------------------------------------------------------------
/*JOBPARM SYSAFF=SYS1
//XCF01    EXEC PGM=AGHCGUTL,REGION=0M,
//         PARM=('1 logstream')                          list all
//*        PARM=('1 logstream "2001/04/30 12:10:54.000"')  list newer
//*        PARM=('6 logstream')                          del. all
//*        PARM=('6 logstream "2001/04/19 13:56:40.000"')  del. older
//*
//STEPLIB  DD DISP=SHR,DSN=hlq.SAGHLOAD
//SYSPRINT DD SYSOUT=*,DCB=(DSORG=PS,RECFM=V,BLKSIZE=121)
//
```

Figure 22. Example of a job using AGHCGUTL to list or delete log records

Figure 23 shows a sample job that uses AGHCGUTL to archive records in a generation data group (GDG).

```
//*------------------------------------------------------------------
//* Utility to archive system logger stream data
//*------------------------------------------------------------------
/*JOBPARM SYSAFF=SYS1
//XCF01    EXEC PGM=AGHCGUTL,REGION=0M,
//*        PARM=('10 logstream "2001/04/30 15:04:35.000"') arch older
//         PARM=('11 logstream "2001/04/30 15:04:44.000"') arch / del
//*
//STEPLIB  DD DISP=SHR,DSN=hlq.SAGHLOAD
//*
//*AGHEGLOG  DD DISP=SHR,DSN=hlq.AGHCL001.LOG2
//AGHEGLOG  DD DSN=hlq.TESTGDG(+1),DISP=(NEW,CATLG),
//             SPACE=(CYL,(1,1)),UNIT=SYSDA,
//             DCB=hlq.TESTGDG.MODEL
//*
//SYSPRINT DD SYSOUT=*,DCB=(DSORG=PS,RECFM=V,BLKSIZE=121)
//
```

Figure 23. Example of a job using AGHCGUTL to archive records in a GDG

Figure 24 shows a sample of a job you can use to create a GDG in which to store archive data. Each time you run AGHCGUTL to archive records, a new member of the GDG will be created; the data in the first member will not be overwritten until the number of members reaches the number specified for the LIMIT parameter.

```
//*
//* This example demonstrates GDG I/O
//*------------------------------------------------------------------
//* Create GDG model
//*------------------------------------------------------------------
//MODEL     EXEC PGM=IDCAMS
//DD1       DD DSN=hlq.TESTGDG.MODEL,DISP=(NEW,CATLG),
//             UNIT=SYSDA,SPACE=(TRK,(0)),
//             DCB=(LRECL=32004,BLKSIZE=32008,RECFM=VB)
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
 /* DELETE (hlq.TESTGDG) GDG */
 DEFINE GDG -
        (NAME(hlq.TESTGDG) -
        EMPTY             -
        SCRATCH           -
        LIMIT(255))
/*
```

Figure 24. Sample job to create a GDG

Tracing

MQSeries Adapter writes trace information to the sequential data set allocated by the DD statement AGHEATRC. If there is no data set allocated to AGHEATRC, the trace data is written to the default output device (SYSPRINT).

The trace level set in the TraceLevel parameter of a client or server profile determines what trace information MQSeries Adapter records:

Trace Level	Description
0	Only informational and error messages are written to the trace data set.
1	In addition to the trace entries written for trace level 0, a trace entry is written each time a client or server uses the SetUserID function to change a user ID.
10	In addition to the trace entries written for trace level 1, a trace entry is written each time a client or server function begins or ends.
30	In addition to the trace entries written for trace level 10, the profile parameters and the contents of data areas are written to the trace data set. Note that for an LFT Put command the contents of the file are also traced, and as a result CPU consumption and the size of the resulting trace file can be significant.

The layout of a trace record is shown in Table 8.

Table 8. Layout of a Trace Record

Name	Length (Bytes)	Description
TraceCompId	2	Component ID: **01** MQSeries Adapter client **02** MQSeries Adapter server **03** MQSeries Adapter services
	2	Separator (,)
TraceDate	10	Date in the form **YYYY/MM/DD** where: **YYYY** Year **MM** Month (01 to 12) **DD** Date (01 to 31)
	1	Separator (,)
TraceTime	13	Time in the form **hh:mm:ss.iii** where: **hh** Hour (00 to 23) **mm** Minute (00 to 59) **ss** Second (00 to 59) **iii** Milliseconds (000 to 999)
	2	Separator (,)
TraceModName	8	Name of the module that issued the trace information.
	2	Separator (,)
TraceData	n	Trace data

An example of a trace is shown on page 92.

Table 9. Example of a trace

```
[1]  01, 2001/04/11 08:10:49.771, AGHCACLP,    --> AGHSwLClient::init
     01, 2001/04/11 08:10:49.772, AGHCA051,     --> AGHMQClient::connect
[2]  01, 2001/04/11 08:10:50.002, AGHCA051,     <-- AGHMQClient::connect rc=0
     01, 2001/04/11 08:10:50.002, AGHCA051,     --> AGHMQClient::openSendQueue
     01, 2001/04/11 08:10:50.003, AGHCA051,     <-- AGHMQClient::openSendQueue rc=0
     01, 2001/04/11 08:10:50.003, AGHCA051,     --> AGHMQClient::openReplyQueue
     01, 2001/04/11 08:10:50.003, AGHCA051,     <-- AGHMQClient::openReplyQueue rc=0
     01, 2001/04/11 08:10:50.003, AGHCACLP,    <-- AGHSwLClient::init rc=0
     01, 2001/04/11 08:10:50.003, AGHCACLP,    --> AGHSwLClient::SwCall
     01, 2001/04/11 08:10:50.004, AGHCASC2,     --> AGHSecurity::check
     01, 2001/04/11 08:10:50.045, AGHCASC2,     <-- AGHSecurity::check rc=0
[3]  03, 2001/04/11 08:10:50.053, AGHCCU02, WriteLog                             addr: 0x15796028
[4]  03, 2001/04/11 08:10:50.053, AGHCCU02, 000000  00000060  00000032  C1C7C8F0  00000100  * ...-....AGH0.... *
     03, 2001/04/11 08:10:50.053, AGHCCU02, 000010  F2F0F0F1  61F0F461  F1F140F0  F87AF1F0  * 2001/04/11 08:10 *
     03, 2001/04/11 08:10:50.053, AGHCCU02, 000020  7AF5F04B  F0F4F640  00000002  00000001  * :50.046 ........ *
     03, 2001/04/11 08:10:50.053, AGHCCU02, 000030  00000000  F1F1F1F1  F1F1F1F1  F1F1F1F1  * ....111111111111 *
     03, 2001/04/11 08:10:50.053, AGHCCU02, 000040  D4A8C299  81958388  40404040  D4A8C485  * MyBranch    MyDe *
     03, 2001/04/11 08:10:50.053, AGHCCU02, 000050  97A39485  95A34040  D2D5D540  40404040  * ptment  KNN      *
     03, 2001/04/11 08:10:50.053, AGHCCU02, 000060  C8859393  9640E696  99938440  86999694  * Hello World from *
     03, 2001/04/11 08:10:50.054, AGHCCU02, 000070  40D2D5D5  40D4C3D3  40839389  8595A340  *  KNN MCL client  *
     03, 2001/04/11 08:10:50.054, AGHCCU02, 000080  A59693A4  948540A3  85A2A340  F5F0F0F0  * volume test 5000 *
     03, 2001/04/11 08:10:50.054, AGHCCU02, 000090  F040                                    * 0                *
     01, 2001/04/11 08:10:50.054, AGHCA051,     --> AGHMQClient::putMsg
     01, 2001/04/11 08:10:50.066, AGHCA051,     <-- AGHMQClient::putMsg rc=0
     01, 2001/04/11 08:10:50.066, AGHCACLP,     --> AGHSwLClient::GetResponse
     .
     .
     .
     01, 2001/04/11 08:10:50.743, AGHCACLP,     <-- AGHSwLClient::GetResponse rc=0
     03, 2001/04/11 08:10:50.743, AGHCCU02, WriteLog                             addr: 0x15796028
     03, 2001/04/11 08:10:50.743, AGHCCU02, 000000  00000060  00000017  C1C7C8F0  00000101  * ...-....AGH0.... *
     03, 2001/04/11 08:10:50.744, AGHCCU02, 000010  F2F0F0F1  61F0F461  F1F140F0  F87AF1F0  * 2001/04/11 08:10 *
     03, 2001/04/11 08:10:50.744, AGHCCU02, 000020  7AF5F04B  F7F4F340  00000002  00000001  * :50.743 ........ *
     03, 2001/04/11 08:10:50.744, AGHCCU02, 000030  00000000  F1F1F1F1  F1F1F1F1  F1F1F1F1  * ....111111111111 *
     03, 2001/04/11 08:10:50.744, AGHCCU02, 000040  D4A8C299  81958388  40404040  D4A8C485  * MyBranch    MyDe *
     03, 2001/04/11 08:10:50.744, AGHCCU02, 000050  97A39485  95A34040  D2D5D540  40404040  * ptment  KNN      *
     03, 2001/04/11 08:10:50.744, AGHCCU02, 000060  C8859393  9640C393  898595A3  40868585  * Hello Client fee *
     03, 2001/04/11 08:10:50.744, AGHCCU02, 000070  84828183  927EF1                        * dback=1          *
     01, 2001/04/11 08:10:50.744, AGHCACLP,    <-- AGHSwLClient::SwCall rc=0
```

Notes:

[1] Whenever a function obtains control, it writes a trace entry that begins with a right-pointing arrow (-->) and the name of the function.

[2] Before the function gives control back to the caller, it writes a trace entry that begins with a left-pointing arrow (<--), the name of the function, and the return code of the function.

[3] When a data area is traced, the first line contains the text that indicates the type of the data, and the start address of the traced area.

[4] When a data area is traced, on each subsequent line:

- The first column to the right of the module name contains the offset (in hex) of the data.
- The four columns to the right of the offset contain 16 bytes of data (in hex).
- The column to the right of the data in hex, in between the asterisks (*), contains the trace data in EBCDIC. Non-printable characters are replaced with a period (.).

Appendix A. Messages and codes

The message IDs of messages issued by MQSeries Adapter have the format

AGH*nnnnc*

where:

nnnn Sequential message number

c Message classification:

E Error

I Information only (no error occurred)

Messages

This section lists alphanumerically by message ID the messages that MQSeries Adapter issues.

AGH1001E Null pointer passed for request or response.

Explanation: The SwCall, SwACall, or SwAWait function was called, but the value of the pointer to the request or response was null.

System Action: The message is not processed.

User Response: This indicates an error in the calling application program. Correct that program so that it supplies a valid pointer with the SwCall, SwACall, or SwAWait function.

AGH1004E Attempt to open profile data set specified in DD statement AGHEAPRO failed.

Explanation: The profile data set, which is specified in the DD statement with the name AGHEAPRO, could not be opened.

System Action: The client or server functions cannot continue, and messages are not sent to the SAG.

User Response: Check the name specified in the DD name AGHEAPRO in your JCL:

- For a CICS environment, the DD name AGHEAPRO must be specified in the CICS startup job.
- For an IMS environment, it must be specified in the IMS MPP or BMP job.

AGH1005E Invalid internal state '*state*'.

Explanation: An internal error occurred.

System Action: The client or server functions cannot continue, and messages are not sent to the SAG.

User Response: Contact your IBM service representative.

AGH1006E Attempt to read profile failed.

Explanation: An error occurred while attempting to read the profile.

System Action: The client or server functions cannot continue, and messages are not sent to the SAG.

User Response: Determine the cause of the file read error. In case of I/O errors, check the job log for additional, system-generated error messages.

AGH1007E Label '*label*' not found in profile.

Explanation: The specified label, which was meant to identify a particular client or server, could not be found in the profile.

System Action: The client or server functions cannot continue, and messages are not sent to the SAG.

User Response: Either change the label used by the calling program to one that is in the profile, or add a new entry to the profile and assign it the appropriate label. The label consists of the client or server name followed by a colon, and marks the beginning of the profile parameters for that client or server.

AGH1008E Mandatory profile parameter '*parm*' not found.

Explanation: The indicated profile parameter is mandatory and was not specified in the profile.

System Action: The client or server functions cannot continue, and messages are not sent to the SAG.

© Copyright IBM Corp. 2001

User Response: Add the missing parameter to the profile.

AGH1009E Value for parameter '*parm*' too long, maximum length=*length*.

Explanation: The value specified for the indicated profile parameter exceeds the maximum length.

System Action: The client or server functions cannot continue, and messages are not sent to the SAG.

User Response: Correct the value.

AGH1010E Client name '*parm*' too long, maximum length=*length*.

Explanation: The client name specified in the class constructor or in the setClientName function exceeds the maximum length.

System Action: The request is rejected.

User Response: Correct the specification of the client name in the function call.

AGH1011E Attempt to open trace data set specified in DD statement AGHEATRC failed.

Explanation: The trace data set, which is specified in the DD statement with the name AGHEATRC, could not be opened.

System Action: Trace output is directed to standard output instead of to a trace data set.

User Response: Correct the error. A likely cause is that the DD Name or name of the trace data set in your JCL is incorrect.

AGH1013E Attempt to write trace data set specified in DD statement AGHEATRC failed.

Explanation: An I/O error occurred during writing to the trace data set, which is specified in the DD statement with the name AGHEATRC.

System Action: Trace output is directed to standard output instead of to a trace data set.

User Response: Correct the error. A likely cause is that the trace output data set was too small to contain all the output.

AGH1014E Attempt to write log data set specified in DD statement '*ddname*' failed.

Explanation: An I/O error occurred during writing to the log data set, which is specified in the indicated DD statement. This can happen when processing client or server functions or in the logger utility.

System Action: The processing is stopped.

User Response: Correct the problem with the log file. A likely cause is that the log output data set is too small. Check for an additional message AGH1080E, which indicates a possible system error.

AGH1018E Profile parameter '*parm*' cannot be saved.

Explanation: The indicated profile parameter is not supported in the current release.

System Action: The parameter is ignored, and processing continues.

User Response: Remove this parameter from the profile.

AGH1025E Logger utility function '*function*' not supported.

Explanation: The indicated logger utility function is not supported.

System Action: The logger utility stops processing.

User Response: Correct the EXEC parameter for the logger utility batch run.

AGH1026E Attempt to allocate memory failed, retcode=*retcode*.

Explanation: There was not enough memory to continue processing.

System Action: Processing stops. If client and server functions are affected, no more messages are sent to the SAG. The program AGHCGUTL stops processing.

User Response: Run the job in an environment in which more memory is available. For example, increase the region size in your JOB statement, EXEC statement, or both.

AGH1027E Attempt to open logger archive file AGHEGLOG failed.

Explanation: The logger archive file, which has the name AGHEGLOG, could not be opened.

System Action: Archiving is not performed.

User Response: Correct the error and rerun the logging utility. Most likely, the DD Name or file name of the logger archive file in your JCL is incorrect.

AGH1028E Variable '*var*' of message '*msgid*' has a null pointer.

Explanation: During creation of the indicated error message, one of the variables was not defined and could not be substituted.

System Action: The message is issued without the indicated variable being set.

User Response: This is an internal error. Contact your IBM service representative.

AGH1029E Message parameter number '*number*' too large, maximum value=*maxval*.

Explanation: The substitution number in an error message exceeds the maximum value. An incorrect error message has been defined in the error message table. The error message table is invalid.

System Action: The parameter in the error message is not substituted.

User Response: This is an internal error. Contact your IBM service representative.

AGH1030E Attempt to open message queue '*qname*' failed, MQCC=*cplcode*, MQRC=*rsncode*.

Explanation: The indicated MQSeries message queue could not be opened. The completion code (MQCC) and reason code (MQRC) shown are MQSeries codes.

System Action: If this error occurred while a server request handler was processing a response, the server request handler issues an error message and continues processing the next request. If this error occurred for a called client or server function, the processing of the function is stopped, and no more messages are sent to the SAG.

User Response: Analyze the completion and reason codes and try to correct the error. The codes are described in *MQSeries for OS/390 Messages and Codes*.

AGH1031E Attempt to connect to queue manager '*mqmname*' failed, MQCC=*cplcode*, MQRC=*rsncode*.

Explanation: The attempt to connect to the indicated MQSeries queue manager failed. The completion code (MQCC) and reason code (MQRC) shown are MQSeries codes.

System Action: The processing of the client or server functions is stopped, and no more messages are sent to the SAG.

User Response: Analyze the completion and reason codes and try to correct the error. The codes are described in *MQSeries for OS/390 Messages and Codes*.

AGH1032E Attempt to put message to message queue '*qname*' failed, MQCC=*cplcode*, MQRC=*rsncode*.

Explanation: The attempt to put a message into the indicated MQSeries message queue failed. The completion code (MQCC) and reason code (MQRC) shown are MQSeries codes.

System Action: If this error occurred while a server request handler was processing a response, the server request handler issues an error message and continues processing the next request. If this error occurred for a called client or server function, the processing of the function is stopped, and no more messages are sent to the SAG.

User Response: Analyze the completion and reason codes and try to correct the error. The codes are described in *MQSeries for OS/390 Messages and Codes*.

AGH1033E Attempt to get message from message queue '*qname*' failed, MQCC=*cplcode*, MQRC=*rsncode*.

Explanation: The attempt to get a message from the indicated MQSeries message queue failed. The completion code (MQCC) and reason code (MQRC) shown are MQSeries codes.

System Action: The processing of the function is stopped, and no more messages are sent to the SAG.

User Response: Analyze the completion and reason codes and try to correct the error. The codes are described in *MQSeries for OS/390 Messages and Codes*.

AGH1034E Attempt to commit message by queue manager '*mqmname*' failed, MQCC=*cplcode*, MQRC=*rsncode*.

Explanation: The attempt by the indicated queue manager to commit a message failed. The completion code (MQCC) and reason code (MQRC) shown are MQSeries codes.

System Action: If this error occurred while a server request handler was processing a response, the server request handler issues an error message and continues processing the next request. If this error occurred for a called client or server function, the processing of the function is stopped, and no more messages are sent to the SAG.

User Response: Analyze the completion and reason codes and try to correct the error. The codes are described in *MQSeries for OS/390 Messages and Codes*.

AGH1035E Attempt to close message queue '*qname*' failed, MQCC=*cplcode*, MQRC=*rsncode*.

Explanation: The attempt to close the indicated MQSeries message queue failed. The completion code (MQCC) and reason code (MQRC) shown are MQSeries codes.

System Action: If this error occurred while a server request handler was processing a response, the server request handler issues an error message and continues processing the next request. If this error occurred for a called client or server function, the processing of the function is stopped, and no more messages are sent to the SAG.

User Response: Analyze the completion and reason

codes and try to correct the error. The codes are described in *MQSeries for OS/390 Messages and Codes*.

AGH1045E Exception in getProfile function.

Explanation: An exception occurred in the getProfile function.

System Action: The profile could not be processed completely. The client or server functions are terminated, and no messages are sent to the SAG.

User Response: Analyze the additional error messages, which indicate the cause of the error.

AGH1048E Profile item '*item*' too long, maximum length=*maxlength*.

Explanation: The specified item in the profile is too long.

System Action: The item is truncated to the indicated maximum length.

User Response: Correct the length of the specified item.

AGH1050E Logging requested, but no DD name or logger stream name specified in profile.

Explanation: Logging was requested (that is, the parameter Log=Y was specified in the profile), but neither a LogDDName nor a LogStreamName was specified in the profile, so logging cannot be done.

System Action: The client or server functions are terminated, and no messages are sent to the SAG.

User Response: Specify a LogDDName or a LogStreamName in the profile and rerun the job.

AGH1051E Attempt to open log data set specified in DD statement '*ddname*' failed.

Explanation: The log data set, which is specified in the indicated DD statement, could not be opened. Logging is not possible.

System Action: This message is preceded by message AGH1080E, which shows the system error message indicating the reason the log file could not be opened. The client and the server functions stop processing, and no more messages are sent to the SAG.

User Response: Analyze the cause of the error as indicated in error message AGH1080E. A likely cause is that a DD statement with the name specified in the LogDDName parameter of the profile is missing from your JCL or is incorrectly specified. Correct the error and rerun the program.

AGH1053E Logger stream name '*lsname*' too long, maximum length=*maxlength*.

Explanation: The length of the indicated logger stream name specified in the profile is too long. Logging is not possible.

System Action: The client and the server functions stop processing, and no more messages are sent to the SAG.

User Response: Correct the specification of the LogStreamName in the profile.

AGH1054E Attempt to connect to logger stream '*lsname*' failed, retcode=*retcode*, rsncode=*rsncode*.

Explanation: A system request to connect to the indicated logger stream failed. Logging is not possible. A likely cause is that the logger stream was not defined in the system. The return and reason codes are described in the *OS/390 MVS Programming: Assembler Services Reference* in the section for the IXGCONN Macro.

System Action: The client and the server functions stop processing, and no more messages are sent to the SAG.

User Response: Analyze the reason code and correct the problem with the logger stream.

AGH1055E Attempt to write to logger stream '*lsname*' failed, retcode=*retcode*, rsncode=*rsncode*.

Explanation: A system request to write to the indicated logger stream failed. Logging is not possible. The return and reason codes are described in the *OS/390 MVS Programming: Assembler Services Reference* in the section for the IXGWRITE Macro.

System Action: The client and the server functions stop processing, and no more messages are sent to the SAG.

User Response: Analyze the reason code and correct the problem with the logger stream, then rerun the logger utility job.

AGH1056E Attempt to browse next entry for logger stream '*lsname*' failed, retcode=*retcode*, rsncode=*rsncode*.

Explanation: A system request to browse next entry in the indicated logger stream failed. This system request occurs only in the logger utility. Logging is not possible. The return and reason codes are described in the *OS/390 MVS Programming: Assembler Services Reference* in the section for the IXGBRWSE Macro.

System Action: The logger utility stops processing.

User Response: Analyze the reason code and correct the problem with the logger stream, then rerun the logger utility job.

AGH1057E Attempt to disconnect logger stream ’*lsname*’ failed, retcode=*retcode*, rsncode=*rsncode*.

Explanation: A system request to disconnect from the indicated logger stream failed. This request occurs during termination only. The return and reason codes are described in the *OS/390 MVS Programming: Assembler Services Reference* in the section for the IXGCONN Macro.

System Action: The logger utility stops processing.

User Response: Analyze the reason code and correct the problem with the logger stream, then rerun the logger utility job.

AGH1058E Attempt to begin browsing entries for logger stream ’*lsname*’ failed, retcode=*retcode*, rsncode=*rsncode*.

Explanation: A system request to browse entries in the indicated logger stream failed. This system request occurs only in the logger utility. Logging is not possible. The return and reason codes are described in the *OS/390 MVS Programming: Assembler Services Reference* in the section for the IXGBRWSE Macro.

System Action: The logger utility stops processing.

User Response: Analyze the reason code and correct the problem with the logger stream, then rerun the logger utility job.

AGH1059E Attempt to delete all entries for logger stream ’*lsname*’ failed, retcode=*retcode*, rsncode=*rsncode*.

Explanation: A system request to delete all entries in the indicated logger stream failed. This system request occurs only in the logger utility. The return and reason codes are described in the *OS/390 MVS Programming: Assembler Services Reference* in the section for the IXGDELET Macro.

System Action: The logger utility stops processing.

User Response: Analyze the reason code and correct the problem with the logger stream, then rerun the logger utility job.

AGH1060E Attempt to delete range of entries for logger stream ’*lsname*’ failed, retcode=*retcode*, rsncode=*rsncode*.

Explanation: A system request to delete entries in the indicated logger stream failed. This system request occurs only in the logger utility. The return code and reason codes are described in the *OS/390 MVS Programming: Assembler Services Reference* in the section for the IXGDELET Macro.

System Action: The logger utility stops processing.

User Response: Analyze the reason code and correct the problem with the logger stream, then rerun the logger utility job.

AGH1061I Logger stream or log data set specified in DD statement ’*logname*’ is empty.

Explanation: The logger utility was called to archive or delete the logger stream or log data set specified in the indicated DD statement, but this logger stream or data set was empty.

System Action: The logger utility stops processing.

User Response: None.

AGH1062E Attempt to end browsing for logger stream ’*lsname*’ failed, retcode=*retcode*, rsncode=*rsncode*.

Explanation: A system request to end the browsing of entries in the indicated logger stream failed. This system request occurs only in the logger utility. Logging is not possible. The return and reason codes are described in the *OS/390 MVS Programming: Assembler Services Reference* in the section for the IXGBRWSE Macro.

System Action: The logger utility stops processing.

User Response: Analyze the reason code and correct the problem with the logger stream, then rerun the logger utility job.

AGH1070E SwCallback function issued return code ’*retcode*’.

Explanation: The SwCallback function issued the indicated return code.

System Action: The server request handler creates an error response and passes it back to the SAG. The format of the error message depends on the format of the retrieved message:

- A basic message is answered with a simple string error message.
- An SNL message is answered with an XML error message.

User Response: The SwCallback function is provided by the application program. Correct the error in the program and rerun it.

AGH1071E No COA received from SAG.

Explanation: The client profile specified that MQSeries is to create confirm-on-arrival (COA) reports. A COA report for a message did not arrive within the time-out

interval. This indicates a problem with the MQSeries connection.

System Action: The client returns the error to the application program.

User Response: Correct the error with the MQSeries connection.

AGH1072E No COD received from SAG.

Explanation: The client profile specified that MQSeries is to create confirm-on-delivery (COD) reports. The COD report is created when the request message is retrieved by the SAG. A COD report for a message did not arrive within the time-out interval. This indicates a problem with the SAG or the MQSeries connection to the SAG.

System Action: The client returns the error to the application program.

User Response: Verify that the MQSeries connection is working, and that the SAG is running.

AGH1073E No response message received from SAG.

Explanation: The response message from the SAG did not arrive within the time-out interval. This indicates a problem with the SAG or with the MQSeries connection to the SAG.

System Action: The client returns the error to the application program.

User Response: Verify that the MQSeries connection is working, and that the SAG is running. You can specify *MQMReports=COA,COD* in the client profile, then rerun the job. MQSeries will then create COA and COD reports that contain information that can help you further isolate the cause of this error.

AGH1074E Message with unexpected MQSeries feedback code received, feedback code=*fbcode*.

Explanation: The client received, in the MQSeries reply-to queue, a report message with the indicated feedback code.

System Action: The message is ignored.

User Response: Verify that no outside applications use the MQSeries reply-to queue.

AGH1075E Unexpected message type received, type=*msgtype*.

Explanation: The client received, in the MQSeries reply-to queue, a message with the indicated message type. This message type is not supported.

System Action: The message is ignored.

User Response: Verify that no outside applications use the MQSeries reply-to queue.

AGH1076E SwCall or LFT failed, retcode=*retcode*.

Explanation: The SwCall or local file transfer (LFT) function failed. The meaning of the return code depends on which component issued this message:

- For a client, the return code indicates whether the SAG returned a message (retcode=224) or did not return a message (retcode=223).
- For a client hub, the return code is the code returned by the SAG.

System Action: The message is not processed, and an answer is not created by the client.

User Response: Analyze the cause of the error and correct it. The trace should contain more detailed error information.

AGH1077E SwAWait cannot be performed before SwACall.

Explanation: The sequence of function calls to the client is incorrect. It does not make sense to wait for a response for a request that has not yet been sent.

System Action: The function call is returned with an error.

User Response: Correct your application program so that it calls the client functions in the proper sequence.

AGH1080E *system-message*

Explanation: A system error occurred. The error message issued by the system (not MQSeries Adapter) is shown here.

System Action: A more specific error message is also issued together with this message.

User Response: See the description of the more specific error message to determine the cause of the error and how to remedy it.

AGH1081I Resource control is not active. Authorization checking will not be performed.

Explanation: One of the following:

- The process user (that is, the user to whom the batch, CICS, or IMS environment belongs) has ALTER access authority for FACILITY(AGH.RS). This access authority disables authorization checking for all messages processed by this user.
- The RACF profile FACILITY(AGH.RS) is not defined. Authorization checking is disabled for all messages and users.

System Action: Messages are processed without authorization checking.

User Response: If this is not what you intend, modify the process user, the process user's authority, or the RACF definitions accordingly.

AGH1082E Authorization check for correspondent name failed, resource class=*class*, resource=*resource*, user ID=*userid*.

Explanation: One of the following:

- The process user (that is, the user to whom the batch, CICS, or IMS environment belongs) does not have READ, UPDATE, or CONTROL access authority for FACILITY(AGH.RS).
- The client user is not authorized to use the indicated resource, which is the correspondent name that was extracted from the client or server profile.

If no user ID is displayed in this message, the user ID is that of the process user.

System Action: The request is not sent to the SAG. For the server request handler function, this error message is passed to the caller and the next message is processed.

User Response: If this message was issued in error, change the RACF definitions so that the user ID is authorized to use the correspondent name.

AGH1083E Authorization check for requestor DN failed, resource class=*class*, resource=*resource*, user ID=*userid*.

Explanation: The indicated user ID is not authorized to use the indicated resource, which is a requestor distinguished name (DN). If no user ID is displayed in this message, the user ID is that of the process user. The requestor DN was extracted from an XML message.

System Action: The request is not sent to the SAG. For the server request handler function, this error message is passed to the caller and the next message is processed.

User Response: If this message was issued in error, change the RACF definitions so that the user ID is authorized to use the DN.

AGH1084E Authorization check for responder DN failed, resource class=*class*, resource=*resource*, user ID=*userid*.

Explanation: The indicated user ID is not authorized to use the indicated resource, which is a responder distinguished name (DN). If no user ID is displayed in this message, the user ID is that of the process user. The responder DN was extracted from an XML message.

System Action: The request is not sent to the SAG. For the server request handler function, this error message is passed to the caller and the next message is processed.

User Response: If this message was issued in error, change the RACF definitions so that the user ID is authorized to use the DN.

AGH1085E Authorization check for signature DN failed, resource class=*class*, resource=*resource*, user ID=*userid*.

Explanation: The indicated user ID is not authorized to use the indicated resource, which is a signature distinguished name (DN). If no user ID is displayed in this message, the user ID is that of the process user. The signature DN was extracted from an XML message.

System Action: The request is not sent to the SAG. For the server request handler function, this error message is passed to the caller and the next message is processed.

User Response: If this message was issued in error, change the RACF definitions so that the user ID is authorized to use the DN.

AGH1090E User ID '*userid*' too long.

Explanation: The indicated user ID, specified in the setUserId function call, is too long. The maximum length is 8 characters.

System Action: The function call returns with an error.

User Response: Correct the application program.

AGH1091E Setting a user ID in a batch job not allowed, process user ID will be used.

Explanation: The setUserId function call was used in the batch environment. For security reasons, this is not allowed.

System Action: The function call returns with an error.

User Response: Correct the application program.

AGH1100E XML parser fatal error at file '*file*', line=*line*, column=*column*.

Explanation: A client or server function tried to use the XML parser to extract information from an XML message in order to check access authorization. The 'file' shown here is the specification given in the DTDFile profile parameter. The XML parser failed at the indicated location. The error message issued by the XML parser is shown in error message AGH1080E.

System Action: Access authorization could not be checked, so the message is not sent to the SAG.

User Response: Analyze the error message issued by the XML parser to find the cause of the error. Most

likely there was a format error in the XML message in the indicated column and line.

AGH1101E The XML parser is not available.

Explanation: A client or server function tried to use the XML parser, but it was not available.

System Action: The request message could not be parsed, so the message is not sent to the SAG.

User Response: Make sure that the XML paser (DLL IXM4C31) is available on your system, and that the library in which it resides is in the STEPLIB DD statement of your JOB. If the parser is not available, download it from http://www-1.ibm.com/servers/eserver/zseries/software/xml/download

---

AGH1102E Error during initialization of XML parser.

Explanation: A client or server function tried to use the XML parser to extract information from an XML message in order to check access authorization. The XML parser could not be initialized. The error message issued by the XML parser is shown in error message AGH1080E.

System Action: Access authorization could not be checked, so the message is not sent to the SAG.

User Response: Analyze the error message issued by the XML parser to find the cause of the error.

AGH1103E Error during parsing of XML document.

Explanation: A client or server function tried to use the XML parser to extract information from an XML message in order to check access authorization. The XML parser failed. The error message issued by the XML parser is shown in error message AGH1080E.

System Action: Access authorization could not be checked, so the message is not sent to the SAG.

User Response: Analyze the error message issued by the XML parser to find the cause of the error. Most likely there is a format error in the XML message.

AGH1110E Client '*clientname*' already active.

Explanation: An application program called the setClientName function for the name indicated, but a client is already active.

System Action: The function call returns an error.

User Response: Correct the application program so that it calls the setClientName function at the correct point in the program flow.

AGH1120E Unrecognizable exception.

Explanation: An unrecognizable exception occurred.

System Action: Normal exception handling is performed. Usually the client or server functions are terminated and no messages are sent to the SAG.

User Response: Usually other error messages precede this error situation; use them to analyze the cause of the error. If necessary, contact your IBM service representative.

AGH1130I Server terminating. Outstanding reports of type *reptype*. MQSeries report ID=*repid*.

Explanation: The server function is terminating, but not all of the expected MQSeries reports of the type indicated (COA, COD, or both) have been received. This can occur when the SAG is not processing the responses in its reply-to queue.

System Action: None.

User Response: Verify that the SAG is working correctly.

AGH1131E Unsupported parameter for MODIFY command: '*parm*'.

Explanation: A MODIFY command was issued with the indicated parameter, but this parameter is not supported. The supported parameters are *stop* and *shut*.

System Action: The input is ignored.

User Response: None.

AGH1132I Request to terminate server request handler or client hub accepted, remaining wait time in seconds: *sec*.

Explanation: A modify command was issued with the *stop* or *shut* parameter for a server request handler or client hub. The server request handler or client hub will terminate immediately after the next request in the queue (for *stop*) or the last request in the queue (for *shut*) is processed. If no requests are in the queue, the server will wait the remaining wait time before terminating.

System Action: The server request handler or client hub stops within the indicated time interval.

User Response: None.

AGH1133E Start of console thread failed, errno=*errno*, console is disabled.

Explanation: A server request handler or client hub tried to start a console interface (*Console=Y* was specified in its profile), but the console interface could not be started. The console interface needs a separate

thread that is started by a pthread_create function call. This function call requires that the run-time option POSIX(ON) be specified. The errno is set by the pthread_create function call.

System Action: The server starts, but not the console interface. To stop the server, terminate the server job using the *cancel* operator command.

User Response: Use the errno to analyze and correct the cause of the error. This code is explained in the *OS/390 C/C++ Run-Time Library Reference*. Most likely, the run-time option POSIX(ON) was not specified, or no OMVS segment is defined in the RACF profile of the process user.

AGH1134E Server function '*function*' called with null pointer.

Explanation: The indicated server function was called with a null pointer as a parameter.

System Action: The requested function is not performed.

User Response: Correct the application program.

AGH1135E Server reply function called before retrieve function.

Explanation: The sequence of function calls to the server is incorrect. You cannot reply to a message before it has been retrieved.

System Action: The message is not processed by the server and not passed to the SAG.

User Response: Correct your application program.

AGH1140E Request '*code*' to convert a message into another code page is not supported.

Explanation: The client and server functions provide an EBCDIC interface to application programs and provide XML messages in UTF-8 encoding to the SAG. This encoding is required by S.W.I.F.T. The client or server functions tried to convert an XML message before it was passed to either an application program or the SAG. The requested conversion could not be performed either because of an internal error or because the message is not in one of the allowed formats.

System Action: The message cannot be processed and is not sent to the SAG.

User Response: Contact your IBM service representative.

AGH1141E Attempt to open code conversion descriptor failed, source=*src*, target=*tgt*.

Explanation: The conversion function uses the ICONV utility. The code conversion descriptor was not available for the indicated source and target code sets. This is most likely due to an installation error. This message is accompanied by message AGH1080E, which displays the system error message.

System Action: The message cannot be processed and is not sent to the SAG.

User Response: Use the information of message AGH1080E to help determine the cause of the error. Ensure the following:

- The correct installation library is accessed for processing of the client or server functions.
- The CEEPrefix parameter in the profile.
- The code pages for the conversion are available in the libraries.

AGH1142E Attempt to convert message from code page '*cp1*' to code page '*cp2*' failed.

Explanation: The conversion function uses the ICONV utility. The code conversion failed. This is an installation or system error. This message is accompanied by message AGH1080E, which displays the system error message.

System Action: The message cannot be processed and is not sent to the SAG.

User Response: Use the information of message AGH1080E to help determine the cause of the error. Ensure that the correct installation library is accessed for processing the client or server functions.

AGH1143E Attempt to convert message to another code page failed.

Explanation: The code conversion utility failed for a client function. This error message is preceded by other error messages that further describe the problem.

System Action: The message cannot be processed and is not sent to the SAG.

User Response: Analyze the accompanying error messages to determine the cause of the error.

AGH1144E Attempt to write XML message to data set specified in DD statement '*ddname*' failed.

Explanation: The current XML message could not be written to the data set specified in the indicated DD statement.

System Action: The operation is terminated, and the XML message is not written.

User Response: None.

AGH9999E Message '*msgID*' not found in message table.

Explanation: An error message with the indicated message identifier should be issued, but is not in the message table (AGHCCMSG.HPP):

- If this chapter contains a description of this message, the message table is not accessible. This is most likely due to an installation error.
- If this chapter does not contain a description of this message, an internal error occurred.

System Action: None.

User Response: Contact your system administrator, who should:

- In case of an installation error, correct the installation
- In case of an internal error, open an APAR

Condition codes

Condition codes issued by clients and servers are shown in the following tables.

Table 10. Condition codes issued by a client

Code	Name	Explanation
0	AGHOK	Operation successful
10	AGHERRORCLIENT	One of the following: • An error occurred while reading the client profile • An error occurred during data conversion • No response could be obtained from SAG • A user error; for example, a null or incorrect pointer was specified
12	AGHERRORMQSeries	MQSeries failed
14	AGHERRORLOGGING	Message logging failed
16	AGHERRORSECURITY	Authorization checking failed; message rejected
18	AGHERRORXML	Error during XML parsing; message rejected
20	AGHERROREXP	Unhandled exception
30	AGHERRORPROFILE	Error while reading profile or mandatory parameter missing

Table 11. Condition codes issued by a server

Code	Name	Explanation
0	AGHOK	Operation successful
14	AGHERRORLOGGING	Message logging failure
16	AGHERRORSECURITY	Security failure, message rejected
18	AGHERRORXML	Error during XML parsing; message rejected
30	AGHERRORPROFILE	Error while reading profile or mandatory parameter missing
32	AGHERRORCONSOLE	Console interface could not be started
34	AGHERRORSEQUENCE	Sequence error; a reply function must be preceded by a retrieve function
36	AGHERRORSWCALLBACK	SwCallback operation failed
38	AGHERRORSWCALL	SwCall operation failed
40	AGHERRORPARM	Parameter is NULL
>2000	MQRC...	MQSeries reason code

SWIFT status codes

SWIFT status codes are issued in response to the following functions:
- SwCall
- SwACall
- SwAWait
- AGHFCall
- AGHLFTCmd
- SwCallback

Table 12. Status codes issued by S.W.I.F.T.

Code	Name	Explanation
0	SwOperationSucceed	The operation was successful.
1	SwOperationFailed	The operation failed. Use the function AGHClientGetConditioncode() to get an additional condition code (see "AGHClientGetConditionCode—Get condition code for client" on page 50).

Return codes

MQSeries Adapter messages can contain the following return codes:

Table 13. Return codes

Return code	Explanation	Module
212	Error during reading the profile, for example the ClientName was wrong or a mandatory parameter was missing.	AGHCACLP
214	The pointer passed from the caller to store the address of the response for the SwCall is a NULL pointer.	
216	The conversion from code page EBCDIC to UTF-8 failed.	
218	The conversion from code page UTF-8 to EBCDIC failed.	
220	Attempt to get storage for the response buffer failed.	
223	Response has zero length (response contains no data). This code is issued only for basic messages (not SNL messages).	
224	Error response received from SAG.	
304	The name of the correspondent was longer than 24 bytes. The name was truncated to 24 bytes. The program will continue.	AGHCASC2
310	The SVC number for third-party authorization checking is not defined in the profile.	
404	More than one element found, first is passed back.	AGHCAXM2
408	Element was not found.	
410	Buffer is not big enough.	

Table 13. Return codes (continued)

Return code	Explanation	Module
510	Open profile failed.	AGHCCU02
512	Read profile failed.	
514	Profile data larger than target field.	
516	Invalid internal state.	
518	The entry for passed key (client/server name) was not found.	
520	Mandatory parameter not found in profile.	
522	Error message not found in message table.	
524	The parameter for the error message is not correct.	
526	Open trace file failed.	
528	Close trace file failed.	
532	Conversion into specified code page is not supported.	
534	Attempt to get storage failed.	
536	Open conversion descriptor failed.	
540	Convert data into another code page failed.	
610	Function not supported.	AGHCGUTL
612	Attempt to get storage failed.	
614	An error occurred while processing the logstream.	
616	Open DD name AGHEGLOG failed.	
618	Write to DD name AGHEGLOG failed.	
704	The source field is bigger than the target field. The value is truncated to the size of the target field.	AGHCG002
734	Attempt to get storage for log record failed.	
2033	No message available; timeout interval elapsed.	AGHCA051
2080	The message buffer was too small to hold the message to be retrieved. MQSeries Adapter continues processing, and tries to obtain a larger buffer.	

Appendix B. Process flows

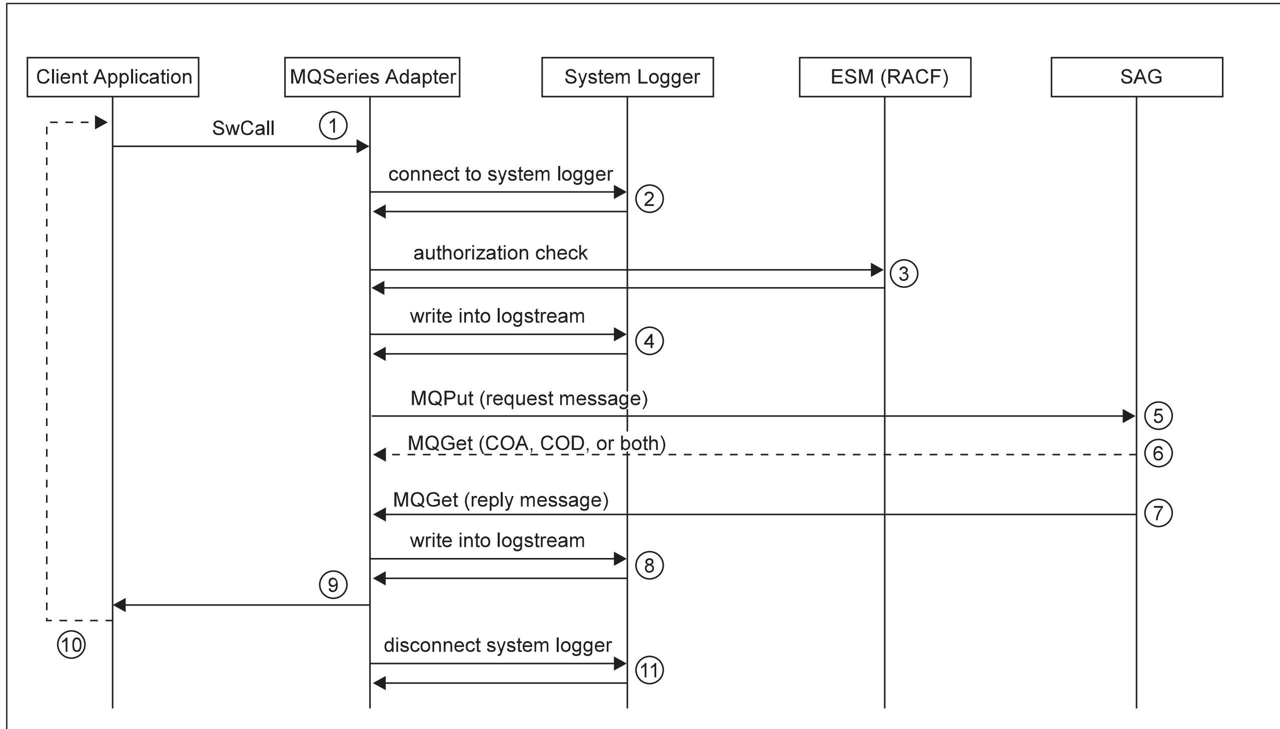

Figure 25. Process flow for a client issuing a synchronous call

Figure 25 shows how the SwCall function is implemented:

1 During the first call to MQSeries Adapter:
- If the DD name for the trace data set was specified, MQSeries Adapter opens this data set.
- If message logging is active and the DD name of a log data set is specified in the client profile, MQSeries Adapter opens this data set.

This step is done only the first time the client is started; if the client runs in a loop, this step is not repeated.

2 If message logging is active and a logger stream name is specified in the client profile, MQSeries Adapter connects to the system logger. This step is done only the first time the client is started; if the client runs in a loop, this step is not repeated.

3 If authorization checking is active, MQSeries Adapter checks whether the user is authorized to access protected resources.

4 If message logging is active, MQSeries Adapter logs the request.

5 MQSeries Adapter puts the request into the request queue.

6 MQSeries Adapter gets the confirmations (COA, COD, or both), if these were requested.

7 MQSeries Adapter gets the response.

© Copyright IBM Corp. 2001

8 If message logging is active, MQSeries Adapter logs the response.

9 MQSeries Adapter returns control to the client.

10 The client can end, or can loop to perform another synchronous call.

11 The client destructor gets control. It closes the trace and message logging data sets, and disconnects from the system logger stream.

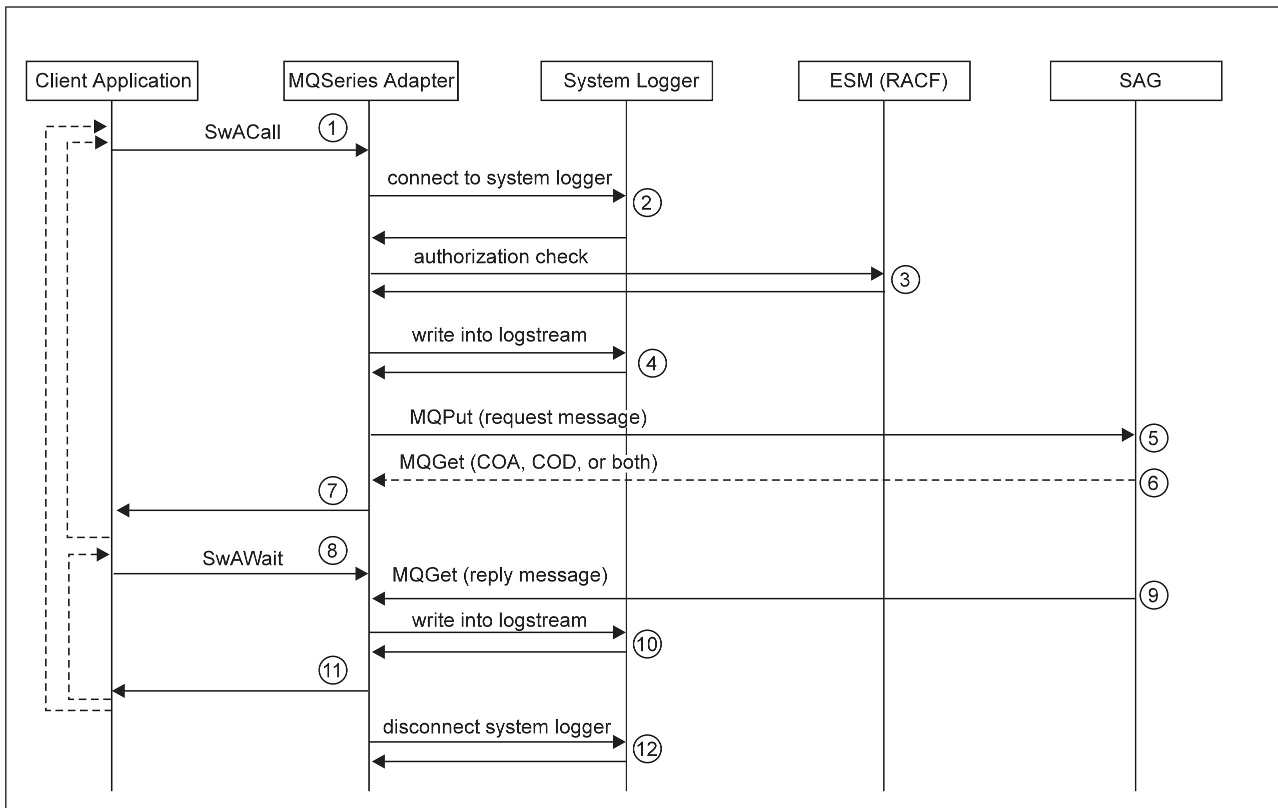

Figure 26. Process flow for a client issuing an asynchronous call

Figure 26 shows how the SwACall and SwAWait functions are implemented:

1 During the first call to MQSeries Adapter:

- If the DD name for the trace data set was specified, MQSeries Adapter opens this data set.
- If message logging is active and the DD name of a log data set is specified in the client profile, MQSeries Adapter opens this data set.

This step is done only the first time the client is started; if the client runs in a loop, this step is not repeated.

2 If message logging is active and a logger stream name is specified in the client profile, MQSeries Adapter connects to the system logger. This step is done only the first time the client is started; if the client runs in a loop, this step is not repeated.

3 If authorization checking is active, MQSeries Adapter checks whether the user is authorized to access protected resources.

4 If message logging is active, MQSeries Adapter logs the request.

5 MQSeries Adapter puts the request into the request queue.

6 MQSeries Adapter gets the confirmations (COA, COD, or both), if these were requested.

7 MQSeries Adapter returns control to the client. The client can continue with other processing, or can loop to perform another asynchronous call.

8 The client calls the SwAWait function.

9 MQSeries Adapter gets the response.

10 If message logging is active, MQSeries Adapter logs the response.

11 MQSeries Adapter returns control to the client. The client can end, or can loop to perform another asynchronous call.

12 The client destructor gets control. It closes the trace and message logging data sets, and disconnects from the system logger stream.

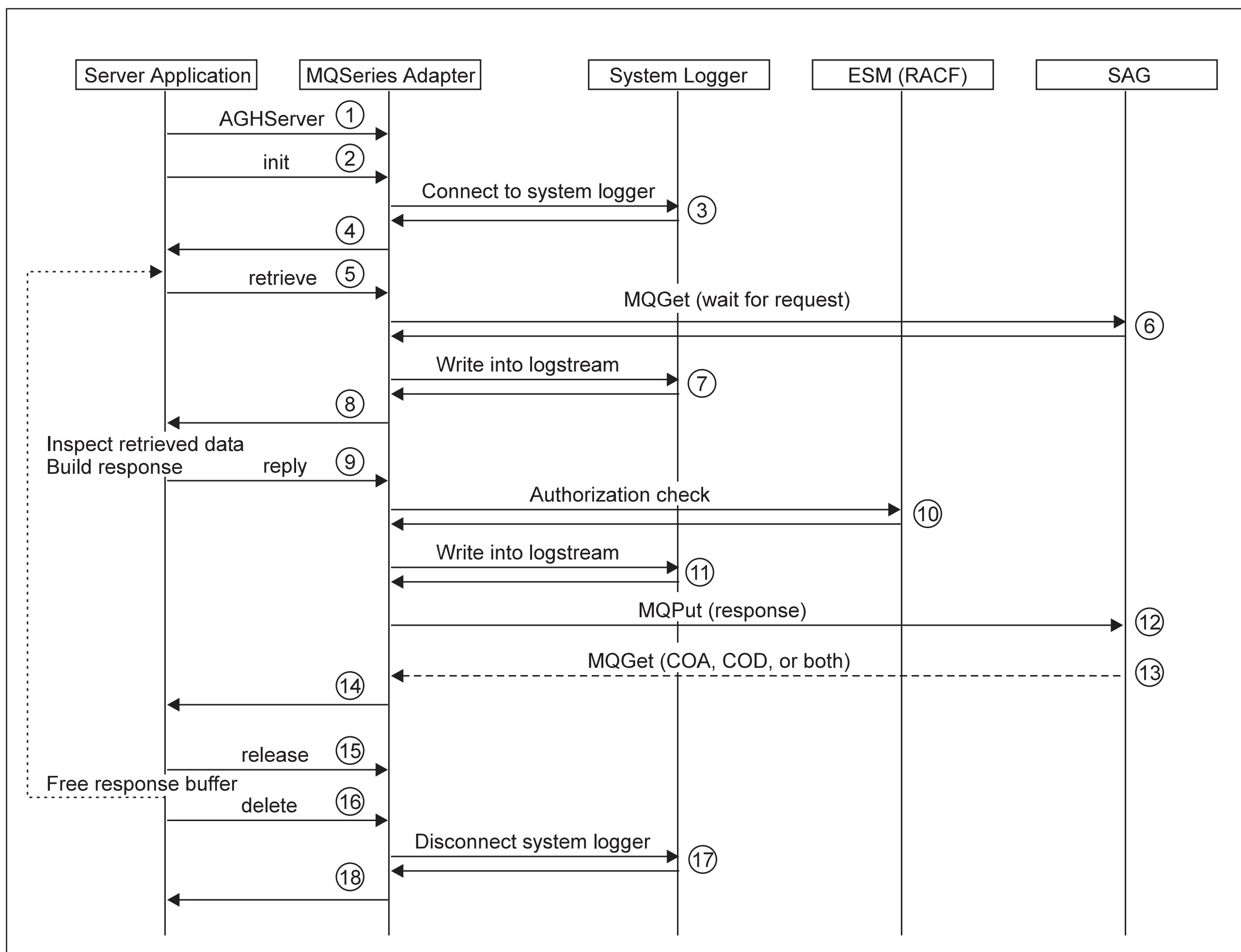

Figure 27. Process flow for a server

Figure 27 shows how to implement a server and how the methods work:

1 Create a new server object.

2 Initialize the server. During this step:

- If the DD name for the trace data set was specified, MQSeries Adapter opens this data set.

- If message logging is active and the DD name of a log data set is specified in the server profile, MQSeries Adapter opens this data set.

3 If message logging is active and a logger stream name is specified in the server profile, MQSeries Adapter connects to the system logger.

4 MQSeries Adapter returns control to the calling application.

5 The application calls the retrieve method.

6 The next message is retrieved from the request queue.

7 If message logging is active, MQSeries Adapter logs the request.

8 The request is passed to the application, which inspects the retrieved data and builds the response.

9 The application calls the reply method.

10 If authorization checking is active, MQSeries Adapter checks whether the user is authorized to access protected resources.

11 If message logging is active, MQSeries Adapter logs the response.

12 MQSeries Adapter puts the response into the reply-to queue.

13 MQSeries Adapter gets the confirmations (COA, COD, or both), if these were requested.

14 MQSeries Adapter returns control to the calling application, which frees the response buffer created by the retrieve method.

15 The application releases the response buffer. The application can end, or can loop to retrieve another request.

16 The application deletes the server object.

17 The server destructor gets control. It closes the trace and message logging data sets, and disconnects from the system logger stream.

18 MQSeries Adapter returns control to the calling application.

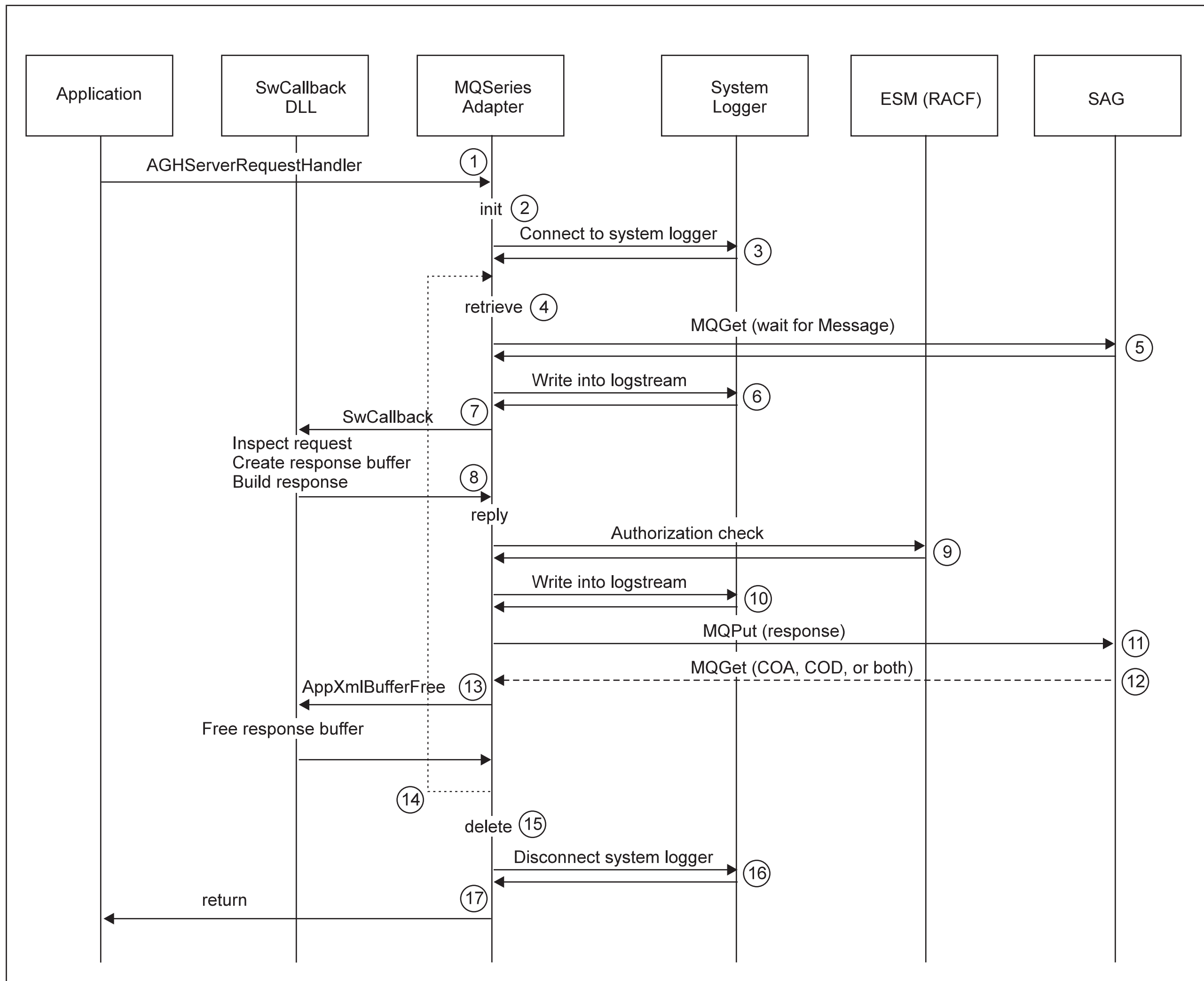

Figure 28. Process flow for a server request handler

Figure 28 shows how to implement a server request handler:

1 The application creates and initializes a new server request handler.

2 During the initialization step:

- If the DD name for the trace data set was specified, MQSeries Adapter opens this data set.
- If message logging is active and the DD name of a log data set is specified in the server profile, MQSeries Adapter opens this data set.

3 If message logging is active and a logger stream name is specified in the server profile, MQSeries Adapter connects to the system logger.

4 The server request handler calls the retrieve method.

5 The next message is retrieved from the request queue.

6 If message logging is active, MQSeries Adapter logs the request.

7 The server request handler calls the SwCallback function. This user-written function typically inspects the request, creates the response buffer, and generates a response.

8 SwCallback returns control to the server request handler, which calls the reply method.

9 If authorization checking is active, MQSeries Adapter checks whether the user is authorized to access protected resources.

10 If message logging is active, MQSeries Adapter logs the response.

11 MQSeries Adapter puts the response into the reply-to queue.

12 MQSeries Adapter gets the confirmations (COA, COD, or both), if these were requested.

13 The server request handler calls the AppXmlBufferFree function, which frees the response buffer and returns control to the server request handler.

14 If there is another request in the request queue, the server request handler retrieves it.

15 If the request queue is empty, the server destructor gets control.

16 The server destructor closes the trace and message logging data sets, and disconnects from the system logger stream.

17 The server request handler returns control to the calling application.

Appendix C. Considerations when implementing authorization checking

The MQSeries Adapter authorization checking services require a type-3 SVC, and the definition of new RACF general resource classes and RACF profiles.

Authorization checking SVC routine

Some of the MQSeries Adapter authorization checking services call RACF services that must run in an authorized environment, for example, in supervisor state. However, programs that call MQSeries Adapter authorization checking services might run in problem program state rather than in supervisor state.

MQSeries Adapter authorization checking services use a type-3 SVC routine. The SVC routine is provided as a load module named **AGHRSSVC**. To enable calling of authorized RACF services, copy this SVC routine into an OS/390 system library under a member name that is associated with the selected SVC number.

The parameters of the SVC must be provided by means of an SVCPARM entry in an SVC table provided by your installation, and located in the SYS1.PARMLIB. SVC tables are members in the SYS1.PARMLIB, and have names of the form **IEASVCxx**, for example IEASVC01. The sample SVCPARM entry for the MQSeries Adapter authorization checking SVC is:

```
SVCPARM 215,REPLACE,TYPE(3),APF(NO)  /* IBM MQSeries Adapter RS SVC  */
```

The load module name of the SVC routine can be specified as another parameter of the SVCPARM entry. This parameter can be omitted if the installation follows the standard OS/390 SVC naming rules.

The standard type-3 SVC name has the form **IGC00*ddx***, where ***dd*** represents the first two digits of the three-digit SVC number, and ***x*** is the letter that corresponds to the last digit: A=1, B=2, C=3, D=4, E=5, F=6, G=7, H=8, I=9. For example, a type-3 SVC with the number 215 would be named IGC0021E.

Type-3 SVC routines are resident in system libraries, for example the Pagable Link Pack Area (PLPA). Load modules that are resident in the PLPA are stored in the load library SYS1.LPALIB. Use the OS/390 Linkage Editor, the batch utility IEBCOPY, or the TSO COPY command to copy the MQSeries Adapter authorization checking SVC routine AGHRSSVC into SYS1.LPALIB.

An SVC table in the SYS1.PARMLIB must be identified by a pointer in the OS/390 system configuration (member IEASYSxx in the SYS1.PARMLIB). The sample pointer to SVC table IEASVC01 in IEASYS01 is:

```
SVC=01          Installation defined SVCs from IEASVC01
```

In order for OS/390 to recognize new or replaced SVCs, an IPL of the OS/390 system may be necessary.

For a detailed description of how to create SVCs, see *OS/390 MVS Programming: Authorized Assembler Services Guide* and *OS/390 MVS Initialization and Tuning Reference*.

© Copyright IBM Corp. 2001

Installation-defined RACF classes

Each of the resources for which MQSeries Adapter can control access is supported by an installation-defined RACF general resource class. The following description contains some hints to help you add installation-defined RACF resource classes to the set of RACF classes provided by IBM. For a more complete description of how to add installation-defined classes to the RACF Class Descriptor Table (CDT), refer to *OS/390 Security Server (RACF) System Programmer's Guide* and *OS/390 Security Server (RACF) Macros and Interfaces*.

RACF class definition source

Installation-defined RACF classes are generated using the RACF macro ICHERCDE. Figure 29 shows the sample ICHERCDE macro instruction used to generate the MQSeries Adapter RACF resource classes.

```
*----------------------------------------------------------------------
*     RACF INSTALLATION DEFINED CDT ENTRIES FOR AGH.RS                -
*----------------------------------------------------------------------
      ICHERCDE CLASS=AGH$CORN,        CORRESPONDENT NAME             *
               DFTRETC=8,                                            *
               FIRST=ANY,OTHER=ANY,                                  *
               ID=197,POSIT=189,                                     *
               MAXLNTH=24
      ICHERCDE CLASS=AGH$RQDN,        REQUESTOR DN                   *
               DFTRETC=8,                                            *
               FIRST=ANY,OTHER=ANY,                                  *
               ID=197,POSIT=189,                                     *
               MAXLNTH=128
      ICHERCDE CLASS=AGH$RSDN,        RESPONDER DN                   *
               DFTRETC=8,                                            *
               FIRST=ANY,OTHER=ANY,                                  *
               ID=197,POSIT=189,                                     *
               MAXLNTH=128
      ICHERCDE CLASS=AGH$SGDN,        SIGN DN                        *
               DFTRETC=8,                                            *
               FIRST=ANY,OTHER=ANY,                                  *
               ID=197,POSIT=189,                                     *
               MAXLNTH=128
```

Figure 29. Sample ICHERCDE macro instruction

Notes:

1. The class names (AGH$CORN, AGH$RQDN, AGH$RSDN, AGH$SGDN) cannot be changed.
2. The default return code (DFTRETC) applies when RACF and the class are active, but a profile does not exist for the resource that is being accessed. DFTRETC=0 means, for example, that a resource is unprotected if no profile exists for this resource.
3. The FIRST and OTHER parameters can be more restrictive than ANY. The actual resource name rules and the used characters may be specified by S.W.I.F.T. The sample FIRST=ANY and OTHER=ANY covers any character combinations.
4. The ID and POSIT numbers of installation-defined classes must be coordinated with the ID and POSIT numbers of existing classes. A CLASSACT command for one class covers all classes with the same POSIT number. The POSIT numbers for the MQSeries Adapter RACF resource classes should therefore be the same.

5. The MAXLNTH parameter specifies the maximum length that a RACF resource name can have (for a description of how RACF resource names are generated, see "Determining which RACF profile controls access to a protected resource" on page 15). The maximum value allowed for this parameter is 246. Because the node-type prefixes in a DN are deleted during transformation into a RACF resource name, MQSeries Adapter might be able to process DNs that are slightly longer than the value specified for MAXLNTH.

Generating the ICHRRCDE

RACF resource classes are defined in the RACF CDT. The CDT has two parts, ICHRRCDX and ICHRRCDE. IBM supplies ICHRRCDX with all IBM-supplied RACF classes. Installation-defined RACF classes are contained in ICHRRCDE.

ICHRRCDE must reside in SYS1.LINKLIB or another APF authorized library in the linklist concatenation. Sample JCL to generate an ICHRRCDE containing the MQSeries Adapter authorization checking classes is shown in Figure 30 on page 116.

```
//xxxxCDT  JOB (xxxx),CLASS=A,REGION=4M,USER=user,
//             MSGCLASS=X,MSGLEVEL=(1,1),TIME=60,NOTIFY=user
//ASSEM1  EXEC PGM=ASMA90,
//         PARM='LIST,OBJECT,XREF(SHORT),NODECK',REGION=6M
//*
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DISP=SHR,DSN=SYS1.MODGEN,VOL=SER=xxxxxx,UNIT=DASD
//         DD DISP=SHR,DSN=SYS1.MACLIB,VOL=SER=xxxxxx,UNIT=DASD
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(20,5))
//SYSUT2   DD UNIT=SYSDA,SPACE=(CYL,(10,1))
//SYSLIN   DD  DSN=&&OBJ,SPACE=(3040,(40,40),,,ROUND),UNIT=VIO,
//             DISP=(MOD,PASS),
//             DCB=(BLKSIZE=3040,LRECL=80,RECFM=FBS,BUFNO=1)
//SYSIN    DD *
*----------------------------------------------------------------------
*     RACF INSTALLATION DEFINED CDT ENTRIES FOR AGH.RS                -
*----------------------------------------------------------------------
      ICHERCDE CLASS=AGH$CORN,         CORRESPONDENT NAME             *
               DFTRETC=8,                                             *
               FIRST=ANY,OTHER=ANY,                                   *
               ID=197,POSIT=189,                                      *
               MAXLNTH=24
      ICHERCDE CLASS=AGH$RQDN,         REQUESTOR DN                   *
               DFTRETC=8,                                             *
               FIRST=ANY,OTHER=ANY,                                   *
               ID=197,POSIT=189,                                      *
               MAXLNTH=128
      ICHERCDE CLASS=AGH$RSDN,         RESPONDER DN                   *
               DFTRETC=8,                                             *
               FIRST=ANY,OTHER=ANY,                                   *
               ID=197,POSIT=189,                                      *
               MAXLNTH=128
      ICHERCDE CLASS=AGH$SGDN,         SIGN DN                        *
               DFTRETC=8,                                             *
               FIRST=ANY,OTHER=ANY,                                   *
               ID=197,POSIT=189,                                      *
               MAXLNTH=128
      ICHERCDE
//*
//LINK     EXEC  PGM=IEWL,REGION=2048K,
//          PARM='NCAL,LIST,LET,XREF,SIZE=(768K,100K)'
//*
//SYSPRINT DD  SYSOUT=*
//SYSLMOD  DD  DSN=SYS1.LINKLIB(ICHRRCDE),DISP=SHR
//SYSIN    DD  DSN=&&OBJ,DISP=(OLD,DELETE)
//SYSLIN   DD *
  INCLUDE SYSIN
  ORDER   AGH$CORN
  ORDER   AGH$RQDN
  ORDER   AGH$RSDN
  ORDER   AGH$SGDN
  ORDER   ICHRRCDE
  NAME    ICHRRCDE(R)
```

Figure 30. Sample JCL to generate an ICHRRCDE

Activating installation-defined classes

Before you can define RACF profiles for installation-defined RACF resource classes, you must activate the classes. A RACF SETR command activates all classes with the same POSIT number. An example of RACF commands to activate the MQSeries Adapter authorization checking classes to handle generic resource profiles is:

```
SETR CLASSACT(AGH$CORN)
SETR GENERIC(AGH$CORN)
```

Any of the RACF classes defined for MQSeries Adapter authorization checking can be specified in these commands. Specifying one class has the desired effect on all classes with the same POSIT number.

Defining AGH.RS

The MQSeries Adapter authorization checking facility is enabled as soon as the profile AGH.RS is defined in the RACF FACILITY class. An example of RACF commands to enable the MQSeries Adapter authorization checking facility is:

```
RDEF FACILITY AGH.RS UACC(NONE)
SETR RACLIST(FACILITY) REFRESH
```

A user associated with a process that uses MQSeries Adapter authorization checking must be permitted to FACILITY(AGH.RS). An example of the RACF commands to let user XXX use MQSeries Adapter authorization checking is:

```
PE   AGH.RS CLASS(FACILITY) ID(XXX) ACCESS(READ)
SETR RACLIST(FACILITY) REFRESH
```

The RACF FACILITY class is raclisted. This is why you must refresh the FACILITY raclist after you modify a FACILITY profile.

Permit a user to access an application resource

The permission of an application user to access a protected resource is defined in RACF resource profiles and their access lists. An example of a RACF command to define a correspondent name is:

```
RDEF AGH$CORN CORR.NAME UACC(NONE)
```

An example of a RACF command to give the user with the ID XXX access to this correspondent name is:

```
PE   CORR.NAME CLASS(AGH$CORN) ID(XXX) ACCESS(READ)
```

RACF supports only uppercase letters in class and resource profile names. Mixed case resource names are folded to uppercase before they are passed to RACF. This means, the MQSeries Adapter authorization checking service is case-insensitive with regard to SWIFTNet resource names.

Appendix D. Notices

This information was developed for products and services offered in the U.S.A. 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
North Castle Drive
Armonk, NY 10504-1785
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.

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 Deutschland
Informationssysteme GmbH
Department 3982
Pascalstrasse 100

© Copyright IBM Corp. 2001

70569 Stuttgart
Germany

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 or any equivalent agreement between us.

The following paragraph does apply to the US 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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:
- CICS
- IBM
- IMS
- MQSeries
- MVS
- OS/390
- RACF

Pentium is a trademark of Intel Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Glossary of terms and abbreviations

This glossary defines terms as they are used in this book. If you do not find the terms you are looking for, refer to the *IBM Dictionary of Computing*, New York: McGraw-Hill, and the *S.W.I.F.T. User Handbook*.

A

API. Application programming interface.

application programming interface (API). An interface that a program can use to use data, functions, or services provided by another program.

C

CCSID. Coded character set identifier.

CICS. Customer Information Control System.

client hub. An abbreviation for **MQSeries hub for client requests**.

client profile. In MQSeries Adapter, an entry in the profile data set that specifies the name of and parameters for a single client.

client user. The user whose ID is assigned to a client, for example so that that user's access authority is used instead of that of the process user.

COA. Confirm on arrival.

COD. Confirm on delivery.

coded character set identifier (CCSID). The identifier of a set of graphic characters and their code point assignments.

commit. In MQSeries, to commit operations is to make the changes on MQSeries queues permanent. After putting one or more messages to a queue, a commit makes them visible to other programs. After getting one or more messages from a queue, a commit permanently deletes them from the queue.

confirm-on-arrival (COA) report. An MQSeries report message type created when a message is placed on that queue. It is created by the queue manager that owns the destination queue.

confirm-on-delivery (COD) report. An MQSeries report message type created when an application retrieves a message from the queue in a way that causes the message to be deleted from the queue. It is created by the queue manager that owns the destination queue.

D

DLL. Dynamic link library.

DN. Distinguished name.

document type definition (DTD). 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.

DTD. Document type definition.

E

ESM. External security manager.

F

FileAct. S.W.I.F.T.'s interactive communication service supporting the exchange of files between parties.

file directory. A directory that determines the root directory for LFT commands. Files referred to by an LFT command must reside in this directory or in one of its subdirectories.

G

GDG. Generation data group.

generation data group. A collection of data sets with the same base name that are kept in chronological order. Each data set is called a generation data set.

H

HFS. Hierarchical file system.

I

IMS. Information Management System.

© Copyright IBM Corp. 2001

InterAct. S.W.I.F.T.'s interactive communication service supporting the exchange of request and response messages between two parties.

J

JCL. Job control language.

L

LFT. Local file transfer.

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

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

M

Message Queue Manager (MQM). An IBM licensed program that provides message queuing services. It is part of the MQSeries set of products.

MQM. Message queue manager.

MQSeries. A family of IBM licensed programs that provides message queuing services.

MQSeries hub for client requests. A program provided by MQSeries Adapter that acts as a single, centralized message processor for clients. It handles the sending of requests and the routing of the responses back to the clients. Abbreviated to *client hub*.

O

OMVS. Open MVS.

P

process user. The user to whom a batch, CICS, or IMS environment belongs.

profile. In MQSeries Adapter, an entry in the profile data set that specifies the name of and parameters for a client or server.

Q

queue. In MQSeries, an object onto which message queuing applications can put messages, and from which they can get messages. A queue is owned and maintained by a queue manager.

queue manager. (1) An MQSeries system program that provides queuing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. See also *local queue manager* and *remote queue manager*. (2) The MQSeries object that defines the attributes of a particular queue manager.

R

RACF. Resource Access Control Facility.

remote queue. In MQSeries, a queue that belongs to a remote queue manager. Programs can put messages on remote queues, but they cannot get messages from remote queues. Contrast with *local queue*.

remote queue manager. In MQSeries, a queue manager is remote to a program if it is not the queue manager to which the program is connected.

reply message. In MQSeries, a type of message used for replies to request messages. Corresponds to a SWIFT response message.

reply-to queue. On the client side, the queue into which a client wants a reply message or report message sent. On the server side, the queue into which a server places a reply message or report message that is to be sent to a client.

report message. In MQSeries, a type of message that gives information about another message. A report message usually indicates that the original message cannot be processed.

request message. In S.W.I.F.T. and MQSeries, a type of message used for requesting a reply from another program.

request queue. On the client side, the queue in which a client places a request message to be sent. On the server side, the queue from which a server retrieves a request message to be processed.

Resource Access Control Facility (RACF). An IBM licensed program that provides for access control by identifying and verifying users to the system, authorizing access to protected resources, logging detected unauthorized attempts to enter the system, and logging detected accesses to protected resources.

response message. In SWIFT, a type of message used for replies to request messages. Corresponds to a *reply message* in MQSeries.

S

SAG. SWIFTAlliance Gateway.

Secure Internet Protocol Network (SIPN). A SWIFT network based on the Internet protocol (IP) and related technologies.

server profile. In MQSeries Adapter, an entry in the profile data set that specifies the name of and parameters for a single server.

server request handler. A program provided by MQSeries Adapter that continually retrieves requests from the request queue at its site and, for each request, calls a user-written function.

SIPN. Secure Internet Protocol Network.

SNL. SWIFTNet Link.

SVC. Supervisor call instruction.

S.W.I.F.T. Society for Worldwide Interbank Financial Telecommunication s.c.

SWIFTAlliance Gateway (SAG). An interface product enabling application-to-application communication using SWIFTNet services.

SWIFT network. The network provided and managed by the Society for Worldwide Interbank Financial Telecommunication s.c.

SWIFTNet Link (SNL). A software product available from S.W.I.F.T. that is needed to access all SWIFTNet services.

SWIFTNet services. S.W.I.F.T.'s IP-based communication services that run on the SIPN.

T

transaction. A specific set of input data that triggers the running of a specific process or job; for example, a message destined for an application program.

U

UNIX System Services (USS). A component of OS/390, formerly called OpenEdition (OE), that creates a UNIX environment that conforms to the XPG4 UNIX 1995 specifications. It provides two open system interfaces on the OS/390 operating system:

- An application program interface (API)
- An interactive shell interface

user identification and verification. The acts of identifying and verifying a RACF-defined user to the system during logon or batch job processing. RACF identifies the user by the user ID and verifies the user by the password or operator identification card supplied during logon processing or the password supplied on a batch JOB statement.

UCS. Universal multi-octet coded character set. There are two complementary standards for UCS:

- Unicode from the Unicode Consortium
- ISO/IRC IS 10646-1 from ISO/IEC

USS. UNIX System Services.

UTF-8. UCS transformation format, 8 bit form. UTF-8 is an encoding of ISO-10646 that is backward compatible with US-ASCII; that is, in which UCS characters are transformed into "ASCII-safe" bytes.

X

XML. An abbreviation for Extensible Markup Language, which is a set of rules for forming semantic tags used to identify the various parts that comprise a multi-part document.

Bibliography

- *SWIFTAlliance Gateway Developer Guide Release 1.2.0*
- *SWIFTAlliance Gateway Interface Specification Release 1.2.0*
- *SWIFTAlliance Gateway MQHA Application Programming Guide Release 1.2.0*
- *SWIFTAlliance Gateway MQHA Installation and Configuration Release 1.2.0*
- *SWIFTNet Link User's Guide*
- *COBOL for OS/390 and VM*, SC26-9049
- *OS/390 MVS: Authorized Assembler Services Guide*, GC28-1763
- *OS/390 MVS Programming: Assembler Services Reference*, GC28-1910
- *OS/390 MVS Initialization and Tuning Reference*, SC28-1752
- *OS/390 Security Server (RACF) System Programmer's Guide*, SC28-1913
- *OS/390 Security Server (RACF) Macros and Interfaces*, SC28-1914
- *OS/390 C/C++ Run-Time Library Reference*, SC28-1663
- *Language Environment for OS/390 Customization*, SC28–1941
- *MQSeries for OS/390 System Management Guide*, SC34-5374
- *MQSeries for OS/390 Messages and Codes*, GC34-5375

© Copyright IBM Corp. 2001

Index

A

access methods 40
administering log data 88
AGHCAAS1 9
AGHCAAS2 9
AGHCG001.HPP 87
AGHCGUTL 88
AGHClient (method) 31
AGHClientGetConditionCode 50
AGHClientGetErrorMessage 51
AGHClientSetClientName 52
AGHClientSetUserId 54
AGHFCall
  C++ 32
  C and COBOL 56
AGHFileHeader 27
AGHLFTCmd
  C++ 33
  C and COBOL 58
AGHLFTCmdParm 26
AGHLoggerService (method) 31, 39
AGHResponse 25
AGHServer (method) 39
AGHServerClientHub 61
AGHServerGetConditionCode 62
AGHServerGetErrorMessage 63
AGHServerInit 64
AGHServerRelease 65
AGHServerReply 67
AGHServerRequestHandler 69
AGHServerRetrieve 70
AGHServerSetUserId 72
AGHServerTerm 74
AppXmlBufferFree 85
archiving log records 88
ASCII 11
asynchronous call
  initiate
    C++ 36
    C and COBOL 75
  retrieve response from
    C++ 37
    C and COBOL 77
asynchronous calls 8
asynchronous InterAct transfer
  C++ 36
  C and COBOL 75
authorization checking 2, 13, 113

B

bank information, specifying 24
basic messages 2
branch information, specifying 24
buffer
  free client
    C++ 39
    C and COBOL 81
  free server 85
buffer, release message 65

C

C++ classes 31
calls 49
calls, synchronous and asynchronous 8
CCSID
  data conversion 13
  parameter 22
CEEPrefix (parameter) 22
central request queue, specifying 24
checking authorization 2, 13, 113
classes, C++ 31
client 1
client buffer, free
  C++ 39
  C and COBOL 81
client constructor 31
client hub 1
client hub, starting
  C++ 40
  C and COBOL 61
  general 9
client hub, stopping 10
client hub message flow 6
client name, set
  C++ 35
  C and COBOL 52
client name, specifying 22
client object, creating 31
client profiles, creating 21
clientHub (method) 40
ClientName (parameter) 22
COA 24
COBOL copybooks 25
COD 24
code page 11
coded character set identifier 13, 22
CodedCharacterSetId field 13
codes
  condition 104
  return 105
  SWIFT status 105
command parameters, LFT 26
condition code, get
  client
    C++ 34
    C and COBOL 50
condition codes 104
Console (parameter) 22
console (using to stop a client hub or server request handler) 10
constructor
  client 31
  server 39
conversion, data 11
Convert (parameter) 22
copybooks, COBOL 25
correspondent name 13
  get method 41
  set method 46
correspondent name, specifying 22
CorrespondentName (parameter) 22
create
  client object 31
  profiles 21
  server object 39
customizing 21

D

data conversion 11, 22
data structures 25
deleting log records 88
department information, specifying 24
distinguished name 13
DN 13
document type definition 18, 23
DTD 18, 23
DTDFile (paramter) 23

E

EBCDIC 11
error message, get
  client
    C++ 34
    C and COBOL 51
  server
    C++ 42
    C and COBOL 63
error messages 95
Expiry (parameter) 23
expiry interval, specifying 23

F

file header 27
file queue, specifying 23
file transfer, local 2
FileAct message flow 6
FileAct transfer
  function to initiate 56
  method to initiate 32
financial institution information, specifying 24
flow, process 107
free client buffer
  C++ 39
  C and COBOL 81
free server buffer 85
free server resources 74
function
  AGHClientGetConditionCode 50
  AGHClientGetErrorMessage 51
  AGHClientSetClientName 52
  AGHClientSetUserId 54
  AGHFCall 56
  AGHLFTCmd 58
  AGHServerClientHub 61
  AGHServerGetConditionCode 62
  AGHServerGetErrorMessage 63
  AGHServerInit 64

© Copyright IBM Corp. 2001

function (continued)
AGHServerRelease 65
AGHServerReply 67
AGHServerRequestHandler 69
AGHServerRetrieve 70
AGHServerSetUserId 72
AGHServerTerm 74
AppXmlBufferFree 85
SwACall 75
SwAWait 77
SwCall 79
SwCallback 83
SwXmlBufferFree 81
functions 49

G

get condition code
client
C++ 34
C and COBOL 50
server
C++ 41
C and COBOL 62
get error message
client
C++ 34
C and COBOL 51
server
C++ 42
C and COBOL 63
get logger object
client 35
server 42
get methods 40
getConditionCode (method)
client 34
server 41
getCorrespondentName (method) 41
getErrorMsg (method)
client 34
server 42
getLog 42
getLoggerObj (method)
client 35
server 42
getMqmName (method) 42
getReplyToQueueName (method) 42
getRequestQueueName (method) 43
getServerName (method) 43
getTraceLevel (method) 43
getUserId (method) 43

H

hardware requirements 17

I

init (method) 43
initializing a server 43, 64
installing 17
institution information, specifying 24
InterAct message flow 3
InterAct transfer, asynchronous
C++ 36
InterAct transfer, asynchronous (continued)
C and COBOL 75
InterAct transfer, synchronous
C++ 38
C and COBOL 79

L

LFT command parameters 26
listing log records 88
local file transfer 2
local file transfer command
C++ 33
C and COBOL 58
local file transfer message flow 4
Log (parameter) 24
log data, administering 88
log data set, specifying 24
log record 87
LogDDName (parameter) 24
LogFinBrch (parameter) 24
LogFinDept (parameter) 24
LogFinInst (parameter) 24
logger object, get
client 35
server 42
logger stream, system
administering data in 88
logging, message 87
logging messages 2, 24
LogStreamName 24
loop server 10

M

message, get error
client
C++ 34
server
C++ 42
C and COBOL 63
message buffer, release 65
message descriptor 61
message flow 3
client hub 6
FileAct 6
InterAct 3
local file transfer 4
server request handler 7
message logging 2, 24, 87
messages 95
method
AGHClient 31
AGHFCall 32
AGHLFTCmd 33
AGHLoggerService 31, 39
AGHServer 39
clientHub 40
getConditionCode
client 34
server 41
getCorrespondentName 41
getErrorMsg
client 34
server 42
method (continued)
getLog 42
getLoggerObj
client 35
server 42
getMqmName 42
getReplyToQueueName 42
getRequestQueueName 43
getServerName 43
getTraceLevel 43
getUserId 43
init 43
release 44
reply 44
requestHandler 45
retrieve 46
setClientName 35
setCorrespondentName 46
setLog 47
setServerName 47
setTraceLevel 47
setUserId
client 36
server 48
SwACall 36
SwAWait 37
SwCall 38
SwCallback 83
SwXmlBufferFree 39
methods 31, 40
MQM, specifying 24
MQMD 13, 61
MQMFileQ (parameter) 23
MQMName (parameter) 24
MQMReplyToQ (parameter) 24
MQMReports (parameter) 24
MQMRequestFileQ (parameter) 24
MQMRequestQ (parameter) 24
MQMT_APPL_FIRST 7
MQSeries interface for clients 1
MQSeries queue manager, specifying 24

N

Notices 119

O

OMVS segment 10

P

parameters, LFT command 26
parameters, profile 22
CCSID 22
CEEPrefix 22
ClientName 22
Console 22
Convert 22
CorrespondentName 22
DTDFile 23
Expiry 23
Log 24
LogDDName 24
LogFinBrch 24
LogFinDept 24

parameters, profile 22 *(continued)*
LogFinInst 24
LogStreamName 24
MQMFileQ 23
MQMName 24
MQMReplyToQ 24
MQMReports 24
MQMRequestFileQ 24
MQMRequestQ 24
SecSVCNum 23
TimeOut 23
TraceLevel 24
place a response in the reply-to queue
C++ 44
C and COBOL 67
portability 9
POSIX(ON) 10
process flow 107
profile parameters
CCSID 22
CEEPrefix 22
ClientName 22
Console 22
Convert 22
CorrespondentName 22
DTDFile 23
Expiry 23
Log 24
LogDDName 24
LogFinBrch 24
LogFinDept 24
LogFinInst 24
LogStreamName 24
MQMFileQ 23
MQMName 24
MQMReplyToQ 24
MQMReports 24
MQMRequestFileQ 24
MQMRequestQ 24
SecSVCNum 23
TimeOut 23
TraceLevel 24
profiles, creating 21

Q

queue manager, specifying 24

R

RACF classes, installation-defined 114
release (method) 44
release a server 44
release server message buffer 65
reply (method) 44
reply-to queue, response in the
C++ 44
C and COBOL 67
reply-to queue, specifying 24
reports, specifying which to create 24
request, retrieve a
C++ 46
C and COBOL 70
request, serve a 83
request file queue, specifying 24
requestHandler (method) 45
requestor DN 13
requirements
hardware 17
software 17
resources, free server 74
responder DN 13
response data structure 25
response from an asynchronous call
C++ 37
C and COBOL 77
response in the reply-to queue
C++ 44
C and COBOL 67
retrieve (method) 46
retrieve a request
C++ 46
C and COBOL 70
retrieve response from asynchronous call
C++ 37
C and COBOL 77
return codes 105

S

SAG file queue, specifying 23
sample programs 29
SecSVCNum (parameter) 23
security 13
sequential log data set, specifying 24
serve a request 83
server 1
constructor 39
initializing 43, 64
loop 10
release 44
server, terminate 74
server buffer, free 85
server message buffer, release 65
server object, creating 39
server profiles, creating 21
server request handler
function to start 69
method to start 45
server request handler, starting 9
server request handler, stopping 10
server request handler message flow 7
set client name
C++ 35
C and COBOL 52
set client user ID 54
set methods 40
set user ID
client (C++) 36
server (C++) 48
server (C and COBOL) 72
setClientName 52
setClientName (method) 35
setCorrespondentName (method) 46
setLog (method) 47
setServerName (method) 47
setTraceLevel (method) 47
setUserId (method)
client 36
server 48
sign DN 13
SNL messages 2
software requirements 17
starting a client hub
C++ 40
C and COBOL 61
general 9
starting a server request handler
C++ 45
C and COBOL 69
general 9
status codes, SWIFT 105
stopping a client hub or server request handler 10
structures, data 25
SVC number, specifying 23
SVC routine 113
SwACall
C++ 36
C and COBOL 75
SwAWait
C++ 37
C and COBOL 77
SwCall
C++ 38
C and COBOL 79
SwCallback 83
SWIFT status codes 105
SwXmlBufferFree
C++ 39
C and COBOL 81
synchronous calls 8
synchronous InterAct transfer
C++ 38
C and COBOL 79
system console (using to stop a client hub or server request handler) 10
system logger stream
administering data in 88
specifying a 24

T

terminate server 74
TimeOut (parameter) 23
timeout interval, specifying 23
trace level, specifying 24
TraceLevel (parameter) 24
tracing 90
type-3 SVC 113

U

user ID
set client 54
set method for client (C++) 36
set method for server (C++) 48
UTF-8 11

X

XML 2

Readers' Comments — We'd Like to Hear from You

IBM MQSeries® Adapter for Secure Financial Messaging Gateway
Installation and Programming Guide
Version 1 Release 1

Publication No. SH12-6731-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
SH12-6731-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 0446
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5799-GKZ

SH12-6731-00

Spine information:

IBM MQSeries® Adapter for Secure Financial Messaging Gateway

Installation and Programming Guide

Version 1
Release 1