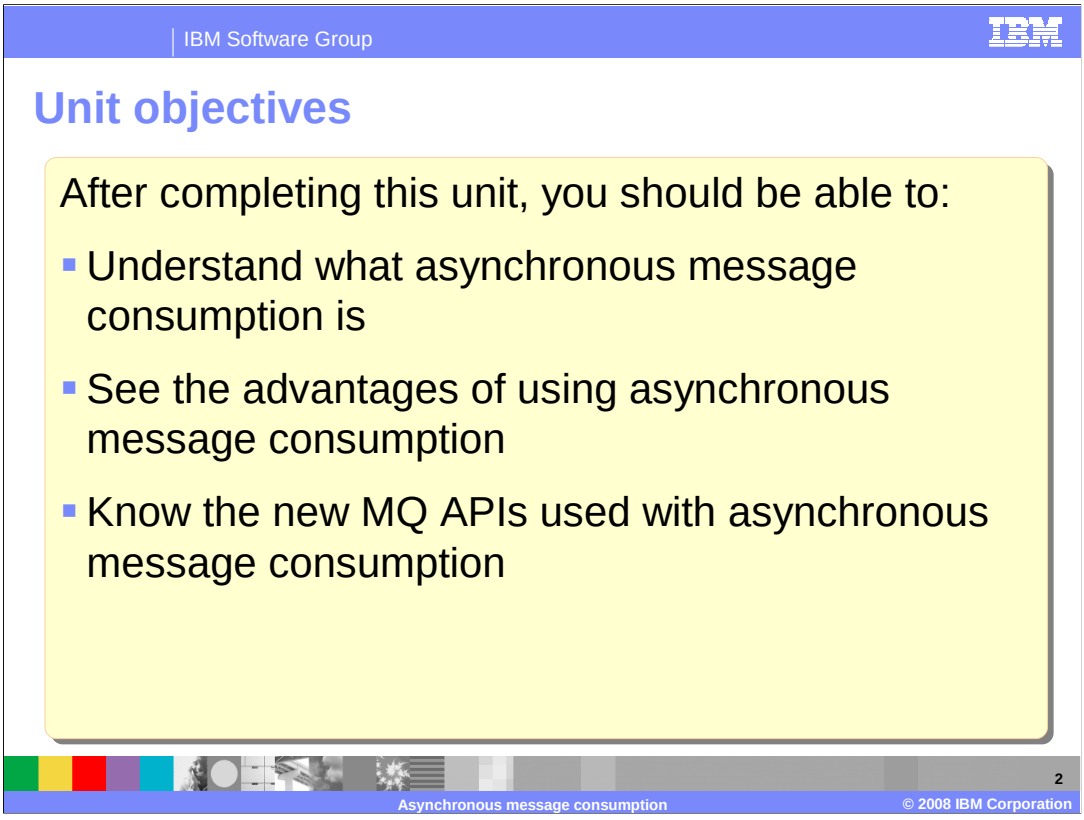
This unit covers the MQ API (or MQI) enhancement in WebSphere MQ version 7.0 covering the new asynchronous message consumption feature added in this release. This unit assumes a reasonable understanding of the existing WebSphere MQ API for putting and getting messages to and from a queue.



After you complete this unit, you should have some understanding of what asynchronous message consumption in MQ version 7 is.

The advantages of asynchronous message consumption in certain circumstances should be clear.

The outline of the new API calls required for asynchronous message consumption should also be understood.

This unit does not attempt to cover the full range of syntax and options available, for which you should refer to the product information center.

What is an asynchronous message consumer

- An asynchronous consumer is a message-driven function or routine, directly invoked by the queue manager to process the messages
- The message data is delivered in a buffer so you normally do not need to do the MQGET by yourself, just process the passed data
- The advantage of this is:
  - Fewer critical resources spent waiting
  - Do not need to allocate a buffer and 'guess' the message size

An asynchronous message consumer is a routine that is initiated by one program. But that then runs independently of the original program consuming and processing messages.

In MQ version 7 the API is extended to allow asynchronous message consumers to be created and managed.

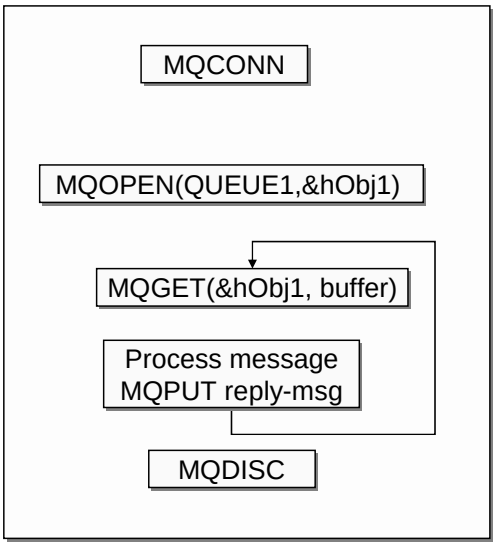
The new API commands are about defining the routines that will be invoked by the queue manager to consume messages from queues and about starting and stopping the message consumption by these routines.

The routines that run asynchronously can use the normal range of MQ API calls, but they do not have to use MQGET to get the message they are processing. That message is passed in a buffer to the routine.

This style of processing can significantly reduce the amount of polling and waiting in applications.

The message consumer routines themselves can be designed to be relatively simple. They can be designed to process the message they receive in a buffer and return.

A typical V6 MQI program processing a single queue



- Simple application design for processing a single queue.
- Polling loop
- With MQGET wait
- Into buffer

Here is a typical version 6 MQ API program processing all the messages from a single queue.

Messages are read from the queue into a buffer supplied by the application program. The buffer size must either be known in advance or the application must first attempt to read the message into a small buffer. If this fails a larger buffer must be allocated and the operation retried.

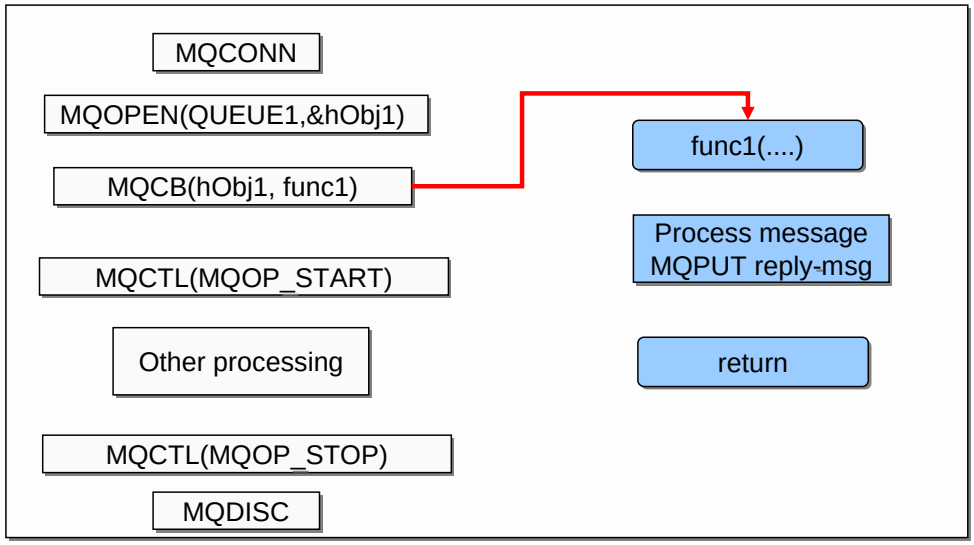
After processing a message and (for example) putting a reply message the program loops to another MQGET, if no message is available the application waits.

If the message arrival rate on the queue is low in relation to the message processing time, this application will spend most of its time waiting for messages.

If the application needs to read messages from more that one queue, a much more complex programming model is required where MQGETS for each queue must be made and a wait issued only when each queue returns no message available. No way exists have the application “wake up” when a message arrives on ANY queue.

Note that on z/OS, the wait with SIGNAL option can be used to achieve this, but only on z/OS.

Asynchronous consumption with V7 MQ API



This slide shows the same application, but this time using an asynchronous message consumption model.

As before the main program uses MQCONN to connect to the queue manager and MQOPEN to open the target queue. Notice this means that MQOPEN can be used with a message selector to limit the messages that will be delivered to the consumers.

This time the new call back registration call (MQCB) is used to register the routine that will process the messages. Here this is a function within the main program but could equally be an external program loaded from a DLL or equivalent.

The MQCB call merely identifies the routine that will be used for processing; it does not in itself start the processing of messages.

The MQ Control (MQCTL) command controls the starting and stopping of all the call back routines registered for this connection handle. The MQOP_START parameter starts the asynchronous processing of messages by the registered routines. In this case the func1 procedure highlighted in blue.

After the MQCTL call to start processing, the main program can continue processing other non MQ work, or can open another connection using MQCONN to do more MQ work. What it must NOT do is use the connection handle that was used in the MQCTL start command – this handle is not available to the main program while asynchronous processing is running.

The main program can eventually stop asynchronous processing with a further MQCTL call as shown and eventually disconnect.

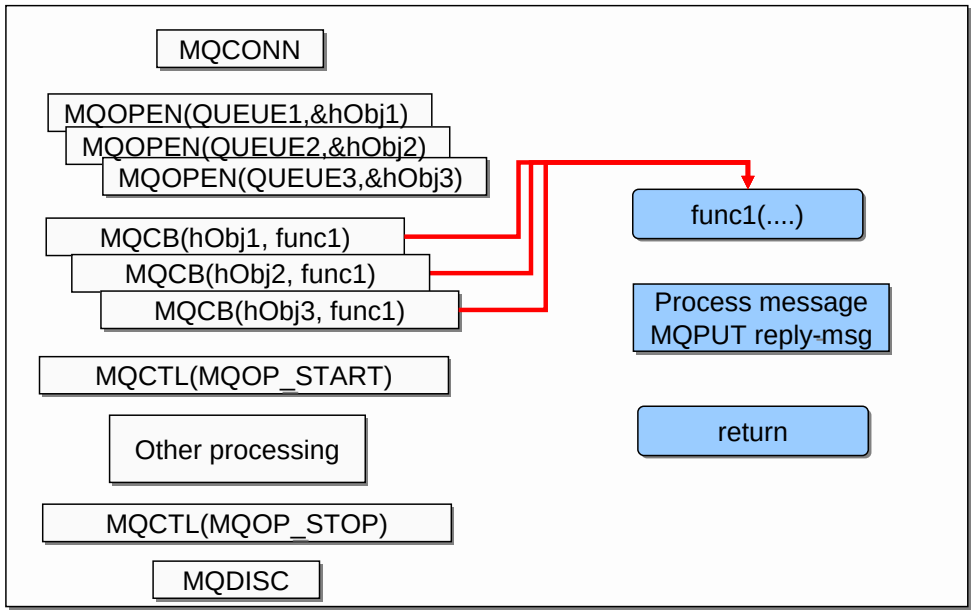
The Asynchronous routine “func1” has a very simple job. It is invoked every time a message is available for processing; the message is presented to the routine in a buffer so no MQGET is required. It has to process the single message to perform any MQ work and return.

The routine can be called for other reasons than message arrival – for example to be notified that the queue is or has become GET INHIBITED, but the main purpose is to process messages.

This design has some clear advantages over the traditional application; first, the processing routine func1 is called only when a message is available. No loop and wait code is required. Second, the main program can continue to do other processing while messages are being consumed. Third, the message buffer is supplied to the application; it does not have to “guess” the maximum size for a message.

The next slide shows how these advantages are multiplied in the case where multiple input queues are being processed.

Asynchronous consumption with multiple queues



This slide shows the code that is required to process input messages from multiple queues, in this case three.

The only differences are that the main program opens all the required queues with MQOPEN and sets up callback routines for each of them, in this case all pointing to the same routine.

The MQCTL processing is the same – this is because the START and STOP applies to all callbacks set up for this connection handle.

Equally the asynchronous processing code func1 does not need modifying at all. Whenever a message is available from any queue, it will be supplied to the routine.

This design is very much simpler than the same design without callback.

New API calls

- MQCB – manage call back
  - A message consumer
    - Registers function to be called when a message, meeting the selection criteria specified, is available on an object handle
  - Event handler
    - Registers a function not related to an object handle to be called when an event, such as a queue manager or connection stopping or quiescing, occurs
- MQCTL – control call back
  - Starts and stops the asynchronous processing for a connection handle

This slide shows the two new calls Manage Call Back and Control Call Back. The Manage call back call (MQCB) is used to register or deregister a function as an asynchronous message consumer. In fact it can register both message consumers, which are routines that are associated with particular queues and are expected to process messages - and event handlers, which are routines that are invoked for more global events such as queue manager stopping or quiescing. This allows cleanup operations to be carried out.

Manage callback is primarily about registering routines to consume messages. Registering the routines does not start them consuming; that is the purpose of the control call back (MQCTL) call. MQCTL can be used to START, STOP, SUSPEND or RESUME asynchronous processing for ALL the registered routines for a particular connection handle.

MQCB – register message consumer

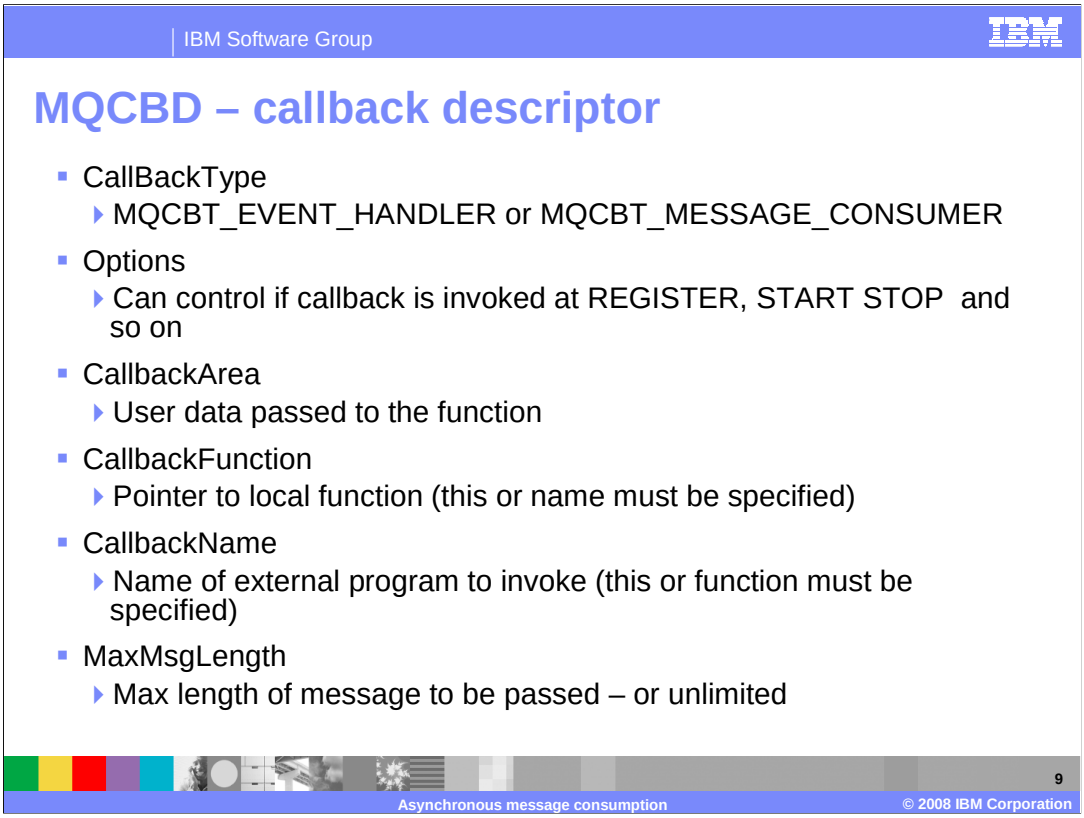
- MQCB (Hconn, operation, CallbackDesc, Hobj, MsgDesc, GetMsgOpts, CompCode, reason)
- Operation
  - MQOP_REGISTER, MQOP_DEREGISTER
  - MQOP_SUSPEND, MQOP_RESUME
- CallbackDesc - MQCBD
  - New structure defining function to be called with options
- MsgDesc and GetMsgOpts
  - Options used to get the message that will be passed to the call back function

This slide looks a little more closely at the register call parameters.

The Operation allows you to REGISTER or DEREGISTER a consumer and to suspend or resume a previously define consumer.

The callback descriptor is covered in the next slide and is used to identify the routine to be called together with additional options.

Message descriptor and Get Message options can also be specified here. These are the options that will be used to get the messages for delivery to the message consumer routine. These can determine, for example, whether messages are to be browsed or consumed. A table later in the presentation shows how some of the options have subtly different meanings in asynchronous message consumption.



Call back descriptor is a new control block used for registering call backs. The callback type specifies either message consumer or event handler.

The Options field is used to indicate when (apart from a message being available) the routine should be called. For example, immediately at registration, when message consumption STARTS and STOPS and so on. This allows initial setup and cleanup to be performed.

The CallbackArea is a user area that is made available to the asynchronous routines. It is not referenced by the queue manager and can contain any application data required.

Either CallbackFunction or CallbackName must be supplied. CallBackFunction is a pointer to a local function to be invoked. CallbackName is a string identifier of an external program to be invoked. Each operating environment has its own way of naming and invoking external programs. In Windows® for example this is the name of an object in a DLL.

MaxMsgLength can be used to limit the size of messages passed to the message consumer; the actual full length is also supplied to the routine

MQCTL – control consumer

- **MQCTL (Hconn, Op, ControlOpts, CompCode, Reason)**
- Op - Operation
  - MQOP_START - Start the consuming of messages for all defined Message Consumer functions for the specified connection handle, application can continue processing other work.
    - MQOP_START_WAIT – as ShTART but this thread waits for all consumers to complete.
  - MQOP_STOP - Stop the consuming of messages and wait for all currently active consumers to complete.
  - MQOP_SUSPEND MQOP_RESUME
- ControlOpts
  - MQCTLO_THREAD_AFFINITY - message consumers for the same connection **must** use the same thread.
  - MQCTLO_FAIL_IF_QUIESCING

The MQCTL call is used to start, stop, suspend and resume ALL the message consumers and event handling routines that have been registered for this connection. The MQCTL with MQOP_START starts all the handlers and flow of control continues in the main program.

From this point the main program can do non-MQ work, or open another connection to process other MQ requests, but should not use the connection handle used in the MQCTL start for any call other than MQCTL with MQOP_STOP or MQOP_SUSPEND (or MQDISC, which performs an implicit STOP).

Note that although the main thread may likely be the task to issue an MQCTL with MQOP_STOP to stop message consumption, this call could be issued by any of the asynchronous consumers. In fact a handler may well terminate message consumption in response to an error situation.

The Start_Wait is a special case where a start is issued but instead of the program continuing, it is suspended until all message consumers have deregistered or suspended or some other thread issues an MQCTL MQOP_STOP or SUSPEND.

The ControlOpts field, and allowing the fail if quiescing option, also allows the message consumers and event handlers to be run in an environment where they are all run on the same thread. Without this option the handlers will run on any thread of the queue managers choosing.

Message consumer and event handler

- When an asynchronous message or event consumer is invoked a fixed parameter set is passed.
  - **MQHCONN** the connection handle
  - **MQMD** message descriptor of the returned message
  - **MQGMO** message options after message get
  - **MQBYTE** message buffer
    - Null for event consumers
  - **MQCBC** call back context – new data area

This slide shows what parameters are passed to a message handler or event consumer. A fixed set of parameters is passed; these are tailored for message handlers and some will be null or meaningless to event handlers.

First is the connection handle required for any MQ calls that the routine may make.

The next two parameters are the message descriptor and the get message options fields with the values that they have after the message has been retrieved from the queue.

The buffer parameter contains the message if one has been returned, and null otherwise.

The last parameter is the MQCBC or Callback Context – a new data area in MQ version 7. This context block contains additional information for the routine.

MQCBC – call back context – key fields

- CallType why function was called
  - Message returned; start, stop, and so on
- CallbackArea user data passed from MQCB
- ConnectionArea user data passed from MQCTL
- CompCode and reason from the MQGET
- State of the consumer
  - Suspended or stopped
- DataLength of the message

Here are some of the key fields from the Callback Context.

CallType contains information about why this function has been called. Two are message delivery call types. MQCBCT_MSG_REMOVED asserts that the message (if any) in the buffer has been destructively removed. MQCBCT_MSG_NOT_REMOVED means that the message has not been removed. These two calltypes will have DataLength set to the amount of data returned. CompCode and ReasonCode should be tested because the message may have been truncated or another failure may have occurred.

Other calltypes do not contain message data and will be delivered if requested in the options field of the call back descriptor when the callback was registered. These include MQCBCT_START_CALL. The purpose of this call type is to allow the callback function to perform some setup when it is started, for example, reinstating resources that were cleaned up when it was previously stopped. The callback function is invoked when the connection is started using either MQOP_START or MQOP_START_WAIT.

CallbackArea is the userdata set in the MQCB call; ConnectionArea is a similar area set in the MQCTL call.

CompCode and Reason relate to the MQGET that got the message.

The state of the consumer will normally be NONE, but can indicate that it is suspended or stopped.

The actual dataLength returned for processing is in the DataLength field.

MQGMO differences

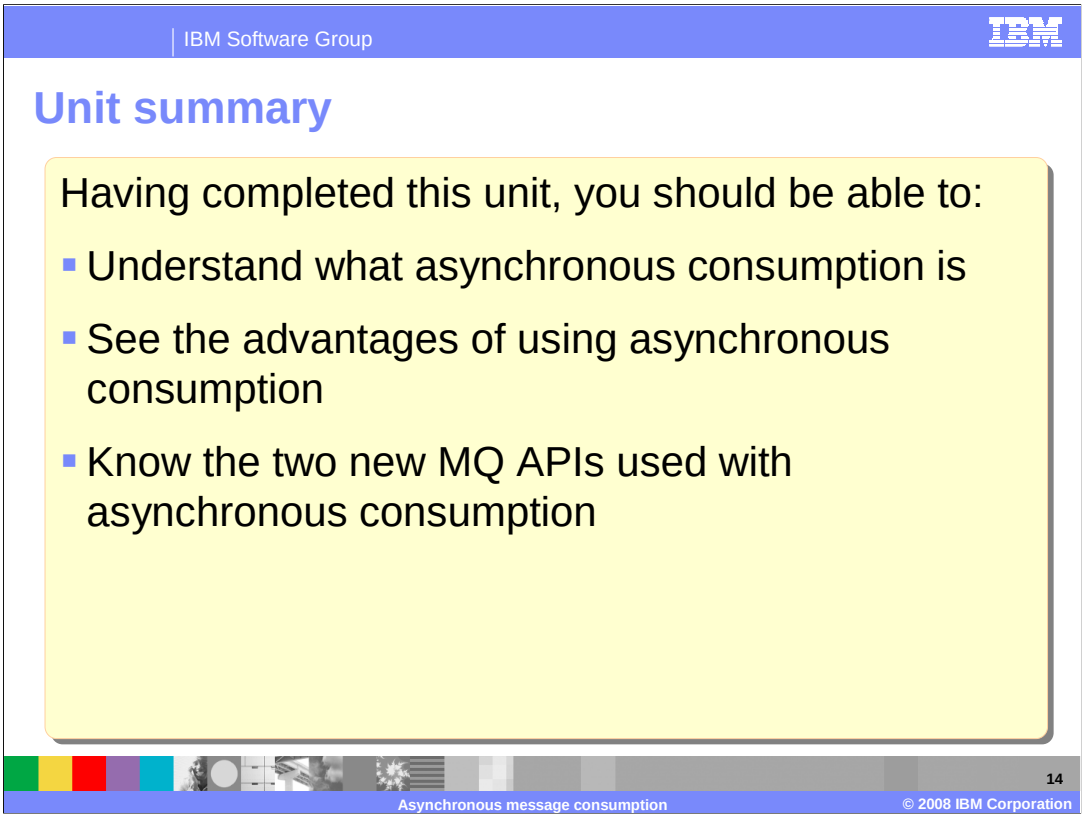
	MQGET call	Asynchronous consumption
MQGMO_BROWSE_FIRST + MQGMO_BROWSE_NEXT	MQRC_OPTIONS_ERROR	Delivers first message then automatically switches to BROWSE_NEXT
MQGMO_WAIT with MQGMO.WaitInterval = 0	MQGET will return immediately with MQRC_NO_MSGS_AVAILABLE if there are no messages	Only called with MQRC_NO_MSGS_AVAILABLE if this is first call or if a message has been returned since last MQRC_NO_MSGS_AVAILABLE.
MQGMO_NO_WAIT		The message consumer will never be called with MQRC_NO_MSGS_AVAILABLE
MQGMO_WAIT with MQGMO.WaitInterval = MQWI_UNLIMITED	MQGET will never return with MQRC_NO_MSGS_AVAILABLE	
MQGMO_SET_SIGNAL	Allowed	Not allowed

The MQCB call provides an MQGMO (get message options) structure that you will be familiar with from using MQGET. The MQGMO is used both for asynchronous consumption and for MQGET. It is used to describe how to consume your message - synchronously or asynchronously. Some of the attributes or options in the MQGMO operate slightly differently when used for asynchronous consumption and this slide details those differences.

Specifying MQGMO_BROWSE_FIRST together with MQGMO_BROWSE_NEXT would be an error in an MQGET Call. But in a message consumer it means read the first message on first invocation – from then on return the next.

MQGMO_WAIT with MQGMO.WaitInterval = 0 operates just like MQGMO_NO_WAIT when you code an MQGET. But in the case of asynchronous consumers NOWAIT means only invoke the routine when messages are available.

In contrast, MQGMO_NO_WAIT and MQGMO_WAIT with a WaitInterval of MQWI_UNLIMITED are quite different when passed to MQGET but in a message consumer their behavior is the same. The consumer will only be passed messages and events; it will never be passed the reason code indicating no messages. Effectively MQGMO_NO_WAIT will be treated as an indefinite wait. This is to prevent the consumer from endlessly being called with the no messages reason code.



Now that you have completed this unit, you should have some understanding of what asynchronous message consumption in MQ version 7 is, and know the advantages of asynchronous message consumption in certain circumstances. The outline of the new API calls required for asynchronous message consumption should also be understood. This unit does not attempt to cover the full range of syntax and options available, for which you should refer to the product information center.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_iea_350_wmqv7_API_5_CallBack.ppt

This module is also available in PDF format at: ../iea_350_wmqv7_API_5_CallBack.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM WebSphere

A current list of other IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.