WebSphere MQ Telemetry Transport
C language implementation

Version 1.2

30 November, 2003

SupportPac author
Ian Harwood

ian_harwood@uk.ibm.com

Take Note!

Before using this report be sure to read the general information under "Notices".

Third Edition, November 2003

This edition applies to Version 1.2 of IA93 and to all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Notices

The following paragraph does not apply in any 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 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.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program that does not infringe any of the intellectual property rights may be used instead of the IBM product.

Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, New York 10594, USA.

The information contained in this document has not be submitted to any formal IBM test and is distributed AS-IS. The use of the information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item has been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Trademarks and service marks

The following terms, used in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

- IBM
- MQSeries
- WebSphere

The following terms are trademarks of other companies:

- Windows, Microsoft

Summary of Amendments

Date	Changes
21 February 2003	Initial release
6 June 2003	Version 1.1 New features: • Bug fixes
30 November 2003	Version 1.2 New features: • Added a persistence interface. • Added DNS resolution so MQIsdp_connect will accept hostnames or IP addresses. • Slight API modifications: o Removed unnecessary version field from CONN_PARMS, SUB_PARMS and UNSUB_PARMS structures. o CONN_PARMS structure now only accepts 1 TCP/IP destination for a broker. It used to accept an array of MQISDP_SVR structures. o MQIsdp_connect has been modified so that Interprocess Communication parameters are passed in a separate MQISDPTI structure instead of in the CONN_PARMS structure. • Tracing has been made visible to the API via a new logLevel parameter in the MQISDPTI structure. • Modified shared library names to contain wmqtt instead of MQIsdp – wmqtt.dll on Windows and libwmqtt.so on UNIX.

Preface

This SupportPac provides a C language implementation of the WMQTT protocol. The code is supplied pre-built for Windows 2000, Linux on i386 and is supplied with a makefile to compile the code on these platforms plus AIX, HP-UX and SUN Solaris. The source code is also supplied to enable the implementation to be modified or ported to other platforms.

WMQTT – WebSphere MQ Telemetry Transport

MQIsdp – MQ Integrator SCADA Device Protocol. This is the former name for the protocol.

Bibliography

- *WebSphere MQ Event Broker 2.1 Programming Guide, IBM Corporation, SC34-6095-00*
- *WebSphere MQ Event Broker 2.1 Introduction and Planning, IBM Corporation , GC34-6088-00*
- *WebSphere Business Integration Message Broker V5.0*
- *WebSphere Business Integration Event Broker V5.0*

Chapter 1. C language WMQTT API and the programming model

The WMQTT protocol is built into a shared library on the WIN32 and UNIX platforms (wmqtt.dll and libwmqtt.so respectively), although the source may be compiled and linked as appropriate for the development platform.

The API provides functions communicating with WebSphere MQ Integrator, such as connecting, disconnecting, publishing, subscribing, unsubscribing, receiving publications and some additional helper functions. The API is designed to be non-blocking, so functions will return before an operation, such as publish or subscribe has completed. The status of these operations can be queried using the message identifier returned by the API.

A timeout value can be specified when receiving publications. A zero timeout value will cause the receive publications to poll to see if any data is available. A greater than zero value will cause the function to efficiently block until either a publication arrives, or the timeout expires.

Any references to WebSphere MQ Integrator broker include the following products:

WebSphere MQ Event Broker V2.1

WebSphere MQ Integrator Broker V2.1

WebSphere MQ Integrator V2.1

WebSphere Business Integration Message Broker V5.0

WebSphere Business Integration Event Broker V5.0

Programming model

The WMQTT C source code may be compiled in one of two ways – to run in a single thread, or 3 threads of execution. The single threaded implementation allows the code to be quickly compiled for evaluation on a platform. The multi-threaded version is considered to be the most desirable, as tasks can be done in the background, such as retrying failed transmissions and keeping the WMQTT connection alive. Obviously the multi-threaded version requires more effort to port.

If using the multi-threaded version then the first thing that must be done is to start up the various threads. See the section below on Single versus Multi-task for more information, as well as the sample applications.

Connecting and disconnecting

When **MQIsdp_connect** returns MQISDP_OK this indicates that a connect message has been successfully built ready to send to the WMQTT broker. The protocol is in a state of CONNECTING.

MQIsdp_status returns the status of the connection between the device and the WMQTT broker, which can be:

- MQISDP_CONNECTING - a connection with the broker is being requested, but no response has been received yet.
- MQISDP_CONNECTED – a response to a connect request has been received, so the protocol is now connected and ready to send data to the broker.
- MQISDP_DISCONNECTED – a TCP/IP error has occurred and the protocol is trying to reconnect to the broker.

- MQISDP_CONNECTION_BROKEN – the protocol has been unable to connect to the broker and all retries have been exhausted, as determined by the retryCount and retryInterval parameters of MQIsdp_connect(). See the documentation for MQIsdp_status for more information.

MQIsdp_disconnect must be called to disconnect the application, even if the connection between the device and the broker is in state MQISDP_CONNECTION_BROKEN. MQIsdp_disconnect frees up resources as well as closing the TCP/IP connection.

Sending data

To send data to the broker the application must use **MQIsdp_publish**. Every piece of data published must be associated with a topic.

Data can be published no matter what state the connection to the broker is in, but applications need to be aware that if the protocol fails to reconnect to the broker after a connection error then the messages will not get delivered. In the event of an error applications can use **MQIsdp_getMsgStatus** to find out what messages have been delivered.

Receiving data

To receive data an application must first tell the broker what data it is interested in receiving. This can be done using **MQIsdp_subscribe** to specify all topics that the application is interested in.

MQIsdp_receivePub can be used to receive data. A timeout can be specified, so that the API blocks until a message arrives, or the timeout expires. MQIsdp_receivePub may return:

- MQISDP_NO_PUBS_AVAILABLE – if there are no publications to receive.
- MQISDP_PUBS_AVAILABLE – if a publication is successfully received and there are more publications available
- MQISDP_OK – if a publication is successfully received and there are no more publications available.
- MQISDP_DATA_TRUNCATED – if there is a message to receive, but the buffer supplied by the application is not large enough.

When an application is no longer interested in receiving data for certain topics it can call **MQIsdp_unsubscribe** specifying all topics for which it no longer wishes to receive data.

The MQISDP_CLEAN_START flag has an affect on subscriptions active within the broker.

If the flag is not specified when connecting then the application must explicitly unsubscribe from all topics, otherwise subscriptions will remain active within the broker even after the application has disconnected. Data will be queued up to send to the application next time it connects.

If the flag is specified then the broker will remove any active subscriptions and outstanding messages when the application disconnects (cleanly or otherwise e.g. a TCP/IP error).

Tracing

Debug output may be produced for all the distinct layers within the protocol implementation. The level of debugging is determined by the logLevel parameter in the MQISDPTI structure that is used to initialize each thread. See the chapter below on Single versus Multi-task for more information about running the protocol in one thread or multiple threads, as well as the sample applications.

Values may be logically OR'ed together to trace more that one aspect of the client at a time. Valid values are:

- LOGNONE – No logging enabled
- LOGNORMAL – Log significant events

- LOGERROR – Log error conditions
- LOGTCPIP – Log TCP/IP i/o events – verbose output
- LOGSCADA – Log WMQTT i/o events
- LOGDEBUG – Produce detailed debug output.

Chapter 2. WMQTT 'C' language API

Connect

```
int MQIsdp_connect( MQISDPCH      *pHconn ,
                    CONN_PARMS * pCp,
                     MQISDPTI      * pTaskInfo );
```

Inputs:

- pHconn - Address of a new connection handle. Its value must be initialised to MQISDP_INV_CONN_HANDLE, otherwise MQISDP_ALREADY_CONNECTED will be returned.
- pCp - Pointer to a CONN_PARMS structure
- pTaskInfo – Pointer to thread specific information. If MQIsdp_StartTasks() is used than this is the structure returned for the API. If MQIsdp_StartTasks() is not used then see Chapter 5.

Returns:

- Return code:
  MQISDP_OK
  MQISDP_NO_WILL_TOPIC
  MQISDP_ALREADY_CONNECTED
  MQISDP_HOSTNAME_NOT_FOUND
  MQISDP_PERSISTENCE_FAILED
  MQISDP_DATA_TOO_BIG
- If return code is MQISDP_OK a valid connection handle is returned otherwise connection handle is set to MQISDP_INV_CONN_HANDLE

CONN_PARMS:

Field	Data Type	Usage
strucLength	long	The length in bytes of the CONN_PARMS structure, including the fixed and variable length portions.
clientId	char[24]	A NULL terminated string up to MQISDP_CLIENT_ID_LENGTH (23) characters in length uniquely identifying the application to the MQIsdp broker.
retryCount	long	Numbers of times to retry a failed connect operation.
retryInterval	long	Interval in seconds at which messages should be retried in a send fails.
options	unsigned short	Options can be combined by using the bitwise OR operation. MQISDP_CLEAN_START : Remove all previous connection history from the broker. See note below. MQISDP_WILL : A Will message is being included, which will be published in the event of the unexpected termination of this application.

		MQISDP_QOS_0 : Quality of Service for the MQISDP_QOS_1 Will message. The highest MQISDP_QOS_2 quality of service specified will be used. MQISDP_WILL_RETAIN : The Will message will be published as a retained publication if this application terminates unexpectedly.
keepAliveTime	unsigned short	A length of time in seconds. If the WMQTT server does not receive any data within this time limit it will assume the application has terminated.
pPersistFuncs	MQISDP_PERSIST*	A pointer to a structure that contains the functions to implement persistence for the protocol. NULL may be specified if persistence is not required.
brokerHostname	char *	A pointer to the hostname or dotted decimal IP address of the broker.
brokerPort	long	The TCP/IP port number of the broker.
Variable length portion of structure		
willTopicLength	Long	Length of the Will topic Only required if option MQISDP_WILL is specified
willTopic	char[n]	The Will topic name Only required if option MQISDP_WILL is specified
willMsgLength	long	Length of the Will message Only required if option MQISDP_WILL is specified
willMsg	char[n]	The Will message data Only required if option MQISDP_WILL is specified

MQISDPTI:

All of this structure except logLevel is populated by MQIsdp_StartTasks(). Only logLevel is used if the code is compiled to run single threaded. For detailed information about the content of this structure see Chapter 5.

Field	**Data type**	**Usage**
apiMailbox	MBH	IPC handle from which the API thread reads
sendMailbox	MBH	IPC handle to which the API writes
receiveMailbox	MBH	Not used

sendMutex	MTH	IPC Mutex coordinating access to the send thread
receiveSemaphore	MSH	IPC semaphore which is signalled when messages arrive to be received.
logLevel	long	The level of logging for the API thread (This must also be set for the MQISDPTI structures for the send and receive threads). Values may be logically OR'ed together and valid values are: • LOGNONE – No logging enabled • LOGNORMAL – Log significant events • LOGERROR – Log error conditions • LOGTCPIP – Log TCP/IP i/o events – verbose output • LOGSCADA – Log WMQTT i/o events • LOGDEBUG – Produce detailed debug output.

- **Note on MQISDP_CLEAN_START:**
  Specifying MQISDP_CLEAN_START means that when an application disconnects cleanly or otherwise (e.g. a TCP/IP error or the unexpected termination of the application) the WMQI broker will clean up on behalf of the application, removing all active subscriptions and any outstanding data for that connection. The WMQTT protocol library will return MQISDP_CONNECTION_BROKEN to the application after the first TCP/IP error.
  If MQISDP_CLEAN_START is not specified then subscriptions and data will remain in the broker in the event of any errors. In this case the protocol library will automatically attempt to reconnect the application in the event of a TCP/IP error. An application will only be returned MQISDP_CONNECTION_BROKEN if the retryCount (as specified in MQIsdp_connect) is exceeded.

Disconnect

int MQIsdp_disconnect(MQISDPCH *pHconn);

Inputs:

- pHconn - Address of a valid connection handle

Returns:

- Return code:
  MQISDP_OK
  MQISDP_PERSISTENCE_FAILED
  MQISDP_CONN_HANDLE_ERROR
- pHconn - Connection handle is set to MQISDP_INV_CONN_HANDLE

To shutdown all threads associated with the protocol call use MQIsdp_terminate(). This simply sets a global variable that causes all threads to exit.
int MQIsdp_terminate(void);

Publish

```
int MQIsdp_publish( MQISDPCH     hConn,
                    MQISDPMH   *pHmsg,
                    PUB_PARMS *pPp );
```

Inputs:

- hConn - A valid connection handle
- pHmsg - Address of a new message handle
- pPp - Pointer to a PUB_PARMS structure

Returns:

- Return code:
  MQISDP_OK
  MQISDP_CONN_HANDLE_ERROR
  MQISDP_Q_FULL
  MQISDP_PERSISTENCE_FAILED
  MQISDP_DATA_TOO_BIG
  MQISDP_CONNECTION_BROKEN
  MQISDP_INVALID_STRUC_LENGTH

- If return code is MQISDP_OK pHmsg points to a valid message handle otherwise it is set to MQISDP_INV_HANDLE. For publications sent at Quality of Service 0 the returned message handle will be zero and cannot be used to query the future state of the publication.

PUB_PARMS:

Field	**Data Type**	**Usage**
strucLength	long	The length in bytes of the PUB_PARMS structure, including the fixed and variable length portions.
options	long	Options can be combined by using the bitwise OR operation. MQISDP_QOS_0 : Quality of Service for the message. The MQISDP_QOS_1 highest quality of service specified MQISDP_QOS_2 will be used. MQISDP_RETAIN : The message will be retained by the WMQTT broker until another publication is received for the same topic.
topicLength	long	The length of the topic
topic	char*	A pointer to the topic to be associated with the data being published
dataLength	long	The length of the data to publish
data	char*	A pointer to the data to be published

Subscribe

```
int MQIsdp_subscribe( MQISDPCH     hConn,
                      MQISDPMH   *pHmsg,
                      SUB_PARMS *pSp );
```

Inputs:

- hConn - A valid connection handle
- pHmsg - Address of a new message handle
- pSp - Pointer to a SUB_PARMS structure

Returns:

- Return code:
  MQISDP_OK
  MQISDP_CONN_HANDLE_ERROR
  MQISDP_Q_FULL
  MQISDP_PERSISTENCE_FAILED
  MQISDP_DATA_TOO_BIG
  MQISDP_CONNECTION_BROKEN
  MQISDP_INVALID_STRUC_LENGTH
- If return code is MQISDP_OK pHmsg points to a valid message handle otherwise it is set to MQISDP_INV_HANDLE

SUB_PARMS:

Field	**Data Type**	**Usage**
strucLength	Long	The length in bytes of the SUB_PARMS structure, including the fixed and variable length portions.
Variable length portion of structure		
topicLength	Long	Length of the topic being subscribed to
topic	char[n]	The name of the topic being subscribed to. The topic name must be 4 byte aligned and padded with space.
options	Long	Options can be combined by using the bitwise OR operation. MQISDP_QOS_0: Quality of Service that data should be published at to this application by the broker. MQISDP_QOS_1 MQISDP_QOS_2

NOTE: topicLength, topic and options must be adjacent and may repeat as a triplet. This will allow an application to subscribe to multiple topics in a single message.

Unsubscribe

```
int MQIsdp_unsubscribe( MQISDPCH      hConn,
                        MQISDPMH      *pHmsg,
                        UNSUB_PARMS *pUp );
```

Inputs:

- hConn - A valid connection handle
- pHmsg - Address of a new message handle

- pUp - Pointer to a UNSUB_PARMS structure

Returns:

- Return code:
  MQISDP_OK
  MQISDP_CONN_HANDLE_ERROR
  MQISDP_Q_FULL
  MQISDP_PERSISTENCE_FAILED
  MQISDP_DATA_TOO_BIG
  MQISDP_CONNECTION_BROKEN
  MQISDP_INVALID_STRUC_LENGTH
- If return code is MQISDP_OK pHmsg points to a valid message handle otherwise it is set to MQISDP_INV_HANDLE

UNSUB_PARMS:

Field	**Data Type**	**Usage**
strucLength	long	The length in bytes of the UNSUB_PARMS structure, including the fixed and variable length portions.
Variable length portion of structure		
topicLength	long	Length of the topic being subscribed to
topic	char[n]	The name of the topic being subscribed to. The topic name must be 4 byte aligned and padded with space.

NOTE: topicLength and topic must be adjacent and may repeat as a pair. This will allow an application to unsubscribe from multiple topics in a single message.

Get Connection Status

```
int MQIsdp_status( MQISDPCH    hConn,
                   long         infoStrLength,
                   long        *pInfoCode,
                   char        *pInfoString );
```

This API call returns the status of the connection between the WMQTT client and the WMQTT broker. This API call does not cause any bytes to be sent across the network.

Inputs:

- hConn - A valid connection handle
- infoStrLength - Length of supplied buffer into which informational data may be copied
- pInfoString - Pointer to a buffer into which an informational string may be placed. Recommended length is MQISDP_INFO_STRING_LENGTH.

Returns:

- Return status code:
  MQISDP_CONN_HANDLE_ERROR

MQISDP_CONNECTING
MQISDP_CONNECTED
MQISDP_DISCONNECTED
MQISDP_CONNECTION_BROKEN

- If the status is MQISDP_CONNECTED
  pInfoString contains the TCP/IP address and port number to which the WMQTT protocol successfully connected.
- If the status is MQISDP_DISCONNECTED
  - If pInfoCode is MQISDP_KEEP_ALIVE_TIMEOUT
    then pInfoString contains the time that the WMQTT server last responded.
  - If pInfoCode is MQISDP_PROTOCOL_VERSION_ERROR
    then the WMQTT broker cannot support the version of the WMQTT protocol specified.
  - If pInfoCode is MQISDP_CLIENT_ID_ERROR
    then the WMQTT broker rejected the client ID for some reason.
  - If pInfoCode is MQISDP_BROKER_UNAVAILABLE
    then the WMQTT broker rejected the connection because the broker is busy.
  - If pInfoCode is MQISDP_SOCKET_CLOSED
    then the WMQTT broker closed the network connection.
  - Otherwise pInfoCode contains the numeric TCP/IP error that occurred and pInfoString indicates whether a send or receive of data failed.

- If the status is MQISDP_CONECTION_BROKEN then the protocol has exhausted attempts to establish a connection with the broker. The retryCount and retryInterval parameters of MQIsdp_connect determine how many attempts are made.
  An application only need worry about reconnecting when a MQIsdp_publish, MQIsdp_subscribe or MQIsdp_unsubscribe fails with MQISDP_CONNECTION_BROKEN, otherwise the WMQTT protocol will keep retrying on behalf of the application to establish a connection.
  When a connection is broken an application can use MQIsdp_getMsgStatus to find out if messages it has sent have been successfully delivered or not.
  The application must then call MQIsdp_disconnect so that resources are freed.

Get Message Status

```
int MQIsdp_getMsgStatus( MQISDPCH  hConn,
                         MQISDPMH hMsg );
```

This API call returns the status of the message being delivered to the WMQTT broker. This API call does not cause any bytes to be sent across the network.

Inputs:

- hConn - A valid connection handle
- hMsg - A valid message handle

Returns:

- Return status code:
  MQISDP_CONN_HANDLE_ERROR
  MQISDP_MSG_HANDLE_ERROR
  MQISDP_DELIVERED
  MQISDP_RETRYING
  MQISDP_IN_PROGRESS

MQISDP_DELIVERED is the final state than a message can get into. A message is delivered once all the Quality of Service WMQTT protocol flows are complete. Messages with a QoS of 0 will be

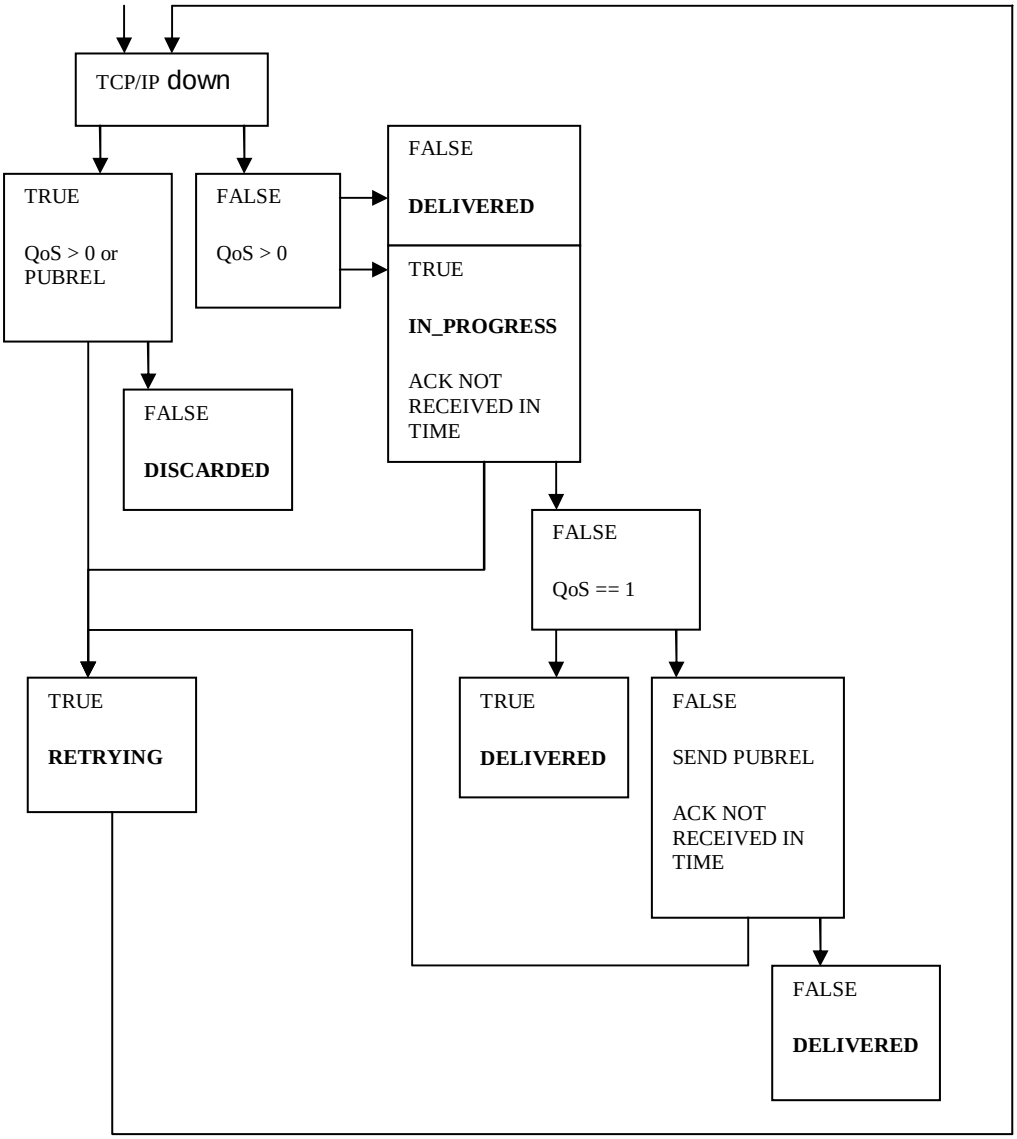
discarded if the TCP/IP connection is down. The application cannot query the state of a publication sent at QoS 0 because the protocol does not know if delivery is successful or not.

MQISDP_MSG_HANDLE_ERROR is returned if an invalid message handle is supplied.

Message State Diagram

The following state diagram shows how the combination of Quality of Service (QoS) and WMQTT message type determine how a message is handled. An understanding of the WMQTT protocol will help understand the diagram.

The first line in each box (TRUE or FALSE) is the result of the test in the previous box. The last item in the box is the test for deciding which the next box to move to is. Any text in bold is the message state that would be returned if MQIsdp_getMsgStatus() were called. This is the state that will be returned until the next block of bold text is encountered.



Receive Publication

```
int MQIsdp_receivePub( MQISDPCH   hConn,
                       long       msTimeout,
```

```
long        *pOptions,
long        *pTopicLength,
long        *pDataLength,
long         msgBufferLength,
char        *pMsgBuffer );
```

This API call returns the next publication that is available to be received, which is in the same order that the publications are received from the WMQTT broker. This API call does not cause any bytes to be sent across the network.

Inputs:

- hConn - A valid connection handle
- msTimeout - A time in milliseconds to wait efficiently for a publication to arrive.
- msgBufferLength - Amount of space available in pMsgBuffer for receiving messages.
- pMsgBuffer - Pointer to a buffer of length msgBufferLength.

Returns:

- Return code:
  MQISDP_CONN_HANDLE_ERROR
  MQISDP_PUBS_AVAILABLE
  MQISDP_NO_PUBS_AVAILABLE
  MQISDP_DATA_TRUNCATED
  MQISDP_OK
- pOptions - contains a bit mask indicating what options were set on the publication message when it was received. Which options are set can be determined by using the bitwise AND operation with the following options:
  - MQISDP_RETAIN
  - MQISDP_QOS_0
  - MQISDP_QOS_1
  - MQISDP_QOS_2
  - MQISDP_DUPLICATE
- pTopicLength – The length in bytes of the topic
- pDataLength - The length in bytes of the data associated with the topic
- pMsgBuffer - The first pTopicLength bytes of this buffer contain the topic, which is followed by pDataLength bytes of message data.

Successful returns:

- If MQISDP_PUBS_AVAILABLE is returned then the application has successfully received a publication and there are more available.
- If MQISDP_OK is returned then the application has successfully received a publication and there are no more available.
- If MQISDP_NO_PUBS_AVAILABLE is returned then there are no publications available to receive.

Failed returns:

- If MQISDP_DATA_TRUNCATED is returned then the application has supplied a buffer that is too small to receive the data. pDataLength contains the actual length of the data allowing the application to reallocate a buffer of this length and reissue the receive publication. pMsgBuffer is filled up to its length with truncated data.

Version and Copyright

```
void MQIsdp_version( void );
```

This prints out the WMQTT protocol version, SupportPac version and copyright information to stdout.

Return Codes

Return Code	Value	Explanation
MQISDP_OK	0	Success
MQISDP_PROTOCOL_VERSION_ERROR	1001	The WMQTT broker does not support this version of the WMQTT protocol
MQISDP_HOSTNAME_NOT_FOUND	1002	If a hostname is used in the connection parameters then this indicates that DNS resolution of the hostname failed.
MQISDP_Q_FULL	1003	The limit on the amount of data in the process of being delivered has been reached. Space will be freed up as messages are delivered or discarded.
MQISDP_FAILED	1004	Failure
MQISDP_PUBS_AVAILABLE	1005	Publications are available to be received.
MQISDP_NO_PUBS_AVAILABLE	1006	No publications are available to be received.
MQISDP_PERSISTENCE_FAILED	1007	When connecting or sending data the persistence implementation reported an error. Investigate the persistence implementation to resolve the problem.
MQISDP_CONN_HANDLE_ERROR	1008	An invalid connection handle has been specified.
MQISDP_NO_WILL_TOPIC	1010	Option MQISDP_WILL has been supplied on MQIsdp_connect, but there is no Will topic.
MQISDP_INVALID_STRUC_LENGTH	1011	An incorrect length supplied in a structure causes the send task to attempt to read beyond the end of the structure.
MQISDP_DATA_LENGTH_ERROR	1012	The data length parameter of MQIsdp_publish is less than zero.
MQISDP_DATA_TOO_BIG	1013	The data supplied is bigger than the WMQTT protocol can handle
MQISDP_ALREADY_CONNECTED	1014	MQIsdp_connect has been called when a connection already exists for the application.
MQISDP_CONNECTION_BROKEN	1017	All attempts by the WMQTT client to establish a connection with the WMQTT broker have been exhausted. MQIsdp_getMsgStatus, MQIsdp_status can be used to find what messages have been delivered and why the connection failed. MQIsdp_receivePub can receive waiting publications.

		The application must disconnect before it is able to send any more data.
MQISDP_DATA_TRUNCATED	1018	The receive buffer supplied for MQIsdp_receivePub is not big enough for the data.
MQISDP_CLIENT_ID_ERROR	1019	The WMQTT broker refused the connection attempt because of a problem with the client identifier.
MQISDP_BROKER_UNAVAILABLE	1020	The WMQTT broker has refused the connection attempt.
MQISDP_SOCKET_CLOSED	1021	The remote socket was closed unexpectedly terminating communications.
MQISDP_OUT_OF_MEMORY	1022	No more memory can be allocated for handling the API call.

Chapter 3. WMQTT Persistence interface

This describes the persistence interface exposed by this implementation of the WMQTT protocol. Developers may write their own persistence implementations as applicable for their environment. A sample implementation is included in mspfp.c which does persistence to the file system.

Using persistence is optional. If a NULL interface is used then the protocol will run entirely in memory, but will not be protected against unexpected application termination.

The persistence is used to store messages being sent from the client to the broker and messages being received from the broker by the client. The sent and received messages should be handled as two separate flows of data as the keys used for persisting the messages are unique in the context of send or receive, but may not be unique across both flows.

The persistence interface function prototypes

At various points in the transmission of a WMQTT message from the client to the broker and vice-versa exit points will be called to record state information to persistent storage. The function names used to implement the persistence are not important. What is important are the function signatures – return codes and parameters. The function signatures must correspond to what is expected in structure MQISDP_PERSIST in MQIsdp.h.

Structure MQISDP_PERSIST or NULL is passed in to MQIsdp_connect indicating if persistence is required or not. The structure contains the function entry points that are required for persistence as well as a void* pointer to pUserData. This pUserData pointer is exclusively for use by the persistence and is specific to the particular persistence implementation being used. Prior to calling MQIsdp_connect pUserData should be initialised to point at a block of storage that the persistence implementation can use for storing state information between calls to the functions. pUserData is passed in as the first parameter to each persistence function.

In the sample file system implementation provided in mspfp.c the pUserData pointer is initialised by the getPersistenceInterface() function.

Opening the persistence

int open(void*pUserData, char *pClientId, char*pBroker, int port);

This is the first function that is called during MQIsdp_connect. The client Id, broker and port taken together uniquely identify the persistence store for this client. The implementation should initialize the persistence ready for use.

Inputs:

- pUserData – For storing state. A block of storage that is specific to the persistence implementation being used.
- pClientId – A string containing the client identifier passed into connect.
- pBroker - A string containing the DNS resolved address of the broker.
- Port - The TCP/IP port of the broker.

Returns:

- This must return 0 on success.
- A non-zero return code indicates failure causing MQISDP_PERSISTENCE_FAILED being retuned to the API. No bytes will be sent over TCP/IP as a result.

Resetting persistence

int reset(void *pUserData);

This function will be called on both MQIsdp_connect and MQIsdp_disconnect if option MQISDP_CLEAN_START is specified. Reset means that all persisted messages should be deleted so that the persistence reverts to its original empty state.

Inputs:

- pUserData – Persistence specific storage.

Returns:

- This must return 0 on success.
- A non-zero return code indicates failure causing MQISDP_PERSISTENCE_FAILED being retuned to the API. No bytes will be sent over TCP/IP as a result.

Initialising persistence

There are two functions to do this:

```
int getAllReceivedMessages( void* pUserData, int *numMsgs, MQISDP_PMSG** pMsgs );
int getAllSentMessages( void* pUserData, int *numMsgs, MQISDP_PMSG** pMsgs );
```

These functions are called during MQIsdp_connect if option MQISDP_CLEAN_START is not specified. These functions are used to reload any previously persisted messages.

Inputs:

- pUserData – Persistence specific storage.

Returns:

- This must return 0 on success.
- A non-zero return code indicates failure causing MQISDP_PERSISTENCE_FAILED being retuned to the API. No bytes will be sent over TCP/IP as a result.
- *numMsgs contains the number of messages being reloaded.
- **pMsgs points to an array of MQISDP_PMSG structures and is of dimension *numMsgs. A MQISDP_PMSG structure must be populated for each message to be reloaded. The array pointer *pMsgs may be freed on the next persistence API call.

The MQISDP_PMSG structure

This structure contains 3 fields:

- unsigned long key – The identifier used when the message was persisted.
- int length – The message length.
- char * pWmqttMsg – A pointer to the message being reloaded. The pWmqttMsg pointer must point to malloced storage as it will be automatically freed when the message is successfully delivered.

Closing persistence

```
int close( void *pUserData );
```

Close is the last persistence function to be called in the lifetime of a connection and is called during MQIsdp_disconnect. Close may close all handles and release all storage being used by the persistence.

Inputs:

- pUserData – Persistence specific storage.

Returns:

- This must return 0 on success.
- A non-zero return code indicates failure causing MQISDP_PERSISTENCE_FAILED being retuned to the API.

Handling sent messages

int addSentMessage(void *pUserData, unsigned long key, int msgLength, char *pWmqttMsg);
int updSentMessage(void *pUserData, unsigned long key, int msgLength, char *pWmqttMsg);
int delSentMessage(void *pUserData, unsigned long key);

These 3 functions are used for persisting data when publications or subscriptions are being sent from the client to the broker.

addSentMessage indicates that a new message is being sent and should be persisted.

updSentMessage indicates that a previously persisted message for the key should be replaced with the new byte array. This is used to handle QoS 2 publications where a PUBREL message replaces a previously persisted publication.

delSentMessage indicates that the message persisted with the specified key should be deleted because it has been successfully delivered.

Inputs:

- pUserData – Persistence specific storage
- key - A unique identifier for the message in the sending context. The functions should use key to uniquely identify the message in the persistence.
- msgLength – The length of the byte array being persisted
- pWmqttMsg – The byte array to be persisted. The byte array contains a message in the WMQTT wire format defined in the protocol specification. It may be a publication, PUBREL or subscription.

Returns:

- Must return 0 on success.
- A non-zero return code indicates failure causing MQISDP_PERSISTENCE_FAILED being retuned to the API in the case of a publication or subscription. In the case of a PUBREL failing then the preceding publish will be retried until the PUBREL is successfully persisted. No bytes will be sent over TCP/IP as a result.
  On failure the persistence is assumed to be in the same state as prior to the function call being made.

Handling received messages

int addReceivedMessage(void *pUserData, unsigned long key, int msgLength, char *pWmqttMsg);
int updReceivedMessage(void *pUserData, unsigned long key);
int delReceivedMessage(void *pUserData, unsigned long key);

These 3 functions are used for persisting data when publications are being received by the client from the broker. The last byte of the pWmqttMsg byte array indicates if a message is eligible for release to the application or not. A QoS 1 message will have bit MQISDP_RELEASED already set. For QoS 2 messages updReceivedMessage should logically OR the last byte of the message with MQISDP_RELEASED and re-persist it.

addReceivedMessage indicates that a new message has been received and should be persisted.

updReceivedMessage indicates that a previously persisted message for the key should be marked as being eligible for release to the application. This is used to handle QoS 2 publications where a PUBREL message indicates that the publication is now eligible for release to the application. See above for how to mark a message as being released.

delSentMessage indicates that the message persisted with the specified key should be deleted because it has been successfully received by the application.

Inputs:

- pUserData – Persistence specific storage
- key - A unique identifier for the message in the receiving context. The functions should use key to uniquely identify the message in the persistence.
- msgLength – The length of the byte array being persisted
- pWmqttMsg – The byte array to be persisted. The byte array contains a message in the WMQTT wire format defined in the protocol specification with 1 additional trailing byte indicating if the publication is eligible for release to the application or not.

Returns:

- Must return 0 on success.
- A non-zero return code indicates failure causing the necessary protocol acknowledgments not to get sent to the broker. The broker will keep retrying to send the data until the client acknowledges receipt.
  On failure the persistence is assumed to be in the same state as prior to the function call being made.

Sample Persistence Implementation

File mspfp.c contains a sample implementation of the persistence interface that uses the file system.

Function getPersistenceInterface(<RootDirectory>) will return a MQISDP_PERSIST structure that can be passed into MQIsdp_connect. The <RootDirectory> may contain '/' or '\' file separators as well as WIN32 drive letters (e.g. C:\) if appropriate. Beneath the specified <RootDirectory> a directory structure is created as follows:

<RootDirectory>/<WMQTT Client Id>/<Broker IP address>_<port>/sent
<RootDirectory>/<WMQTT Client Id>/<Broker IP address>_<port>/rcvd

Sent and received messages are persisted into the sent and rcvd directories respectively. One file per message is created with a name equivalent to the message key.
When sending Quality of Service 2 publications the PUBLISH message is placed in a file with the name of the key. The PUBREL message is placed in a file with the name of the key concatenated with the character 'u'. This is to ensure that the PUBREL is written to disk before the PUBLISH is deleted.

This sample implementation has been ported to the Windows and Linux platforms.

Diagnosing errors

If MQIsdp_connect fails with MQISDP_PERSISTENCE_FAILED then the most likely reason is that the file persistence implementation does not have permission to create files or directories beneath the <RootDirectory>. On UNIX the userid under which the persistence is running must have read, write and execute permission on <RootDirectory>.

Chapter 4. Compiling and linking client applications

To compile and link the WMQTT protocol library see MQIsdp_porting.doc

Includes

Applications must include C header file MQIsdp.h which contains the function prototypes, structures and defines all values used by the API.

Linking on Windows 2000

The client API is contained in wmqtt.dll. To use this DLL an application needs to link with wmqtt.lib.

Linking on Linux

An application needs to link with libwmqtt.so which contains the WMQTT protocol.

Chapter 5. Single versus Multi task solution

The protocol library can be compiled in two ways:

1. To run in one thread of execution.
   This requires a C library and a TCP/IP socket interface in order to compile and run. The API also has to be called sufficiently frequently (at least once per keep alive interval) to stop the protocol timing out.
2. To run in three threads of execution.
   This is higher performing, but requires more advanced OS facilities to coordinate the tasks. These facilities are:
   - An Inter Process Communication (IPC) mechanism e.g. maillots or pipes
   - A single Mutex semaphore
   - A single Resource semaphore

   The WMQTT connection is automatically kept alive and the connection handle can be shared between tasks to allow concurrent sending and receiving of data.
   The application must start up the send and receive threads before using the API.

Define MSP_SINGLE_THREAD when compiling the WMQTT shared library code to produce a version of the protocol that will run in one thread, otherwise the code will be compiled to run in multiple threads of execution.

Running the protocol in a single task

When running the protocol in a single task the application simply calls the API and the protocol flows are executed behind the API.

The API is identical to that used when running in a multi-task environment. The only difference is that when connecting the parameters apiMailbox, sendMailbox, sendMutex and receiveSemaphore can be left undefined in the MQISDPTI structure passed into MQIsdp_connect.

Because there is only one thread of execution the TCP/IP socket is only read when the API is called. If the API is not called sufficiently frequently (at least once per keepalive interval) then the protocol will timeout. Also the TCP/IP stream buffer may fill up if the WMQTT broker is sending publications to the application.

Running the protocol in three tasks

When running in a multi-task environment the send and receive tasks must be started prior to using the API.

The send and receive tasks are **MQIsdp_SendTask** and **MQIsdp_ReceiveTask** respectively. These tasks receive data from the network, send protocol flows and manage the TCP/IP connection in the background without blocking the application. When there is no application connected these tasks close the TCP/IP socket and wait efficiently for the next MQIsdp_connect().

On some embedded systems, particularly safety critical systems, dynamic creation of threads and processes is not allowed. Bearing this in mind the supplied code implements a lowest common denominator solution and does not dynamically create threads or processes. Instead it leaves the send and receive tasks to be started as appropriate for the platform and assumes they have been successfully started before the API is called. **MQIsdp_StartTasks** has been supplied in the shared library and it starts MQIsdp_SendTask and MQIsdp_ReceiveTask as threads correctly for the Win32 and Linux platforms. The source for this in mspstart.c maybe enhanced to support other platforms, or a more static method of creating the tasks may be used.

Creating the send and receive tasks

To successfully start the protocol in multiple threads of execution the necessary Inter Process Communication (IPC) objects need to be created. These are three data buffers, one for each thread (MailSlots on Windows and unnamed pipes on UNIX), a mutex to coordinate access to the send thread and a semaphore to signal when messages are available to receive.

MQIsdp_StartTasks

For Windows and Linux MQIsdp_StartTasks creates all necessary IPC these objects and starts the MQIsdp_SendTask and MQIsdp_ReceiveTask functions as threads. After calling MQIsdp_StartTasks the application may then call MQIsdp_Connect(). This function, which is defined in mspstart.c, may be adapted as appropriate for other platforms.

```
int MQIsdp_StartTasks( MQISDPTI *pApiTaskInfo,
                       MQISDPTI *pSendTaskInfo,
                       MQISDPTI *pRcvTaskInfo,
                       char     *pClientId );
```

Inputs

- pApiTaskInfo – A pointer to an uninitialised MQISDPTI structure, except for field logLevel.
- pSendTaskInfo - A pointer to an uninitialised MQISDPTI structure, except for field logLevel.
- pRcvTaskInfo - A pointer to an uninitialised MQISDPTI structure, except for field logLevel.
- pClientId – A string containing the client identifier that this application will use.

See the section on tracing for more information about logLevel.

Returns

- 0 on success, 1 on error
- pApiTaskInfo, pSendTaskInfo and pRcvTaskInfo are correctly populated with data. The contents of pApiTaskInfo should be used to provide the mailbox, mutex and semaphore parameters to MQIsdp_connect.

In detail

If dynamically creating tasks using MQIsdp_StartTasks is not appropriate for your platform, then these are the rules for creating the send and receive tasks. See MQIsdp_porting.doc for more information.

- **Send task -** MQIsdp_SendTask(MQISDPTI *pTaskInfo);
  This takes a MQISDPTI structure (MQISDP Task Info) as a parameter, populated as follows:
  - sendMailbox - IPC handle which the send task reads from
  - receiveMailbox - IPC handle for the send task to write to the receive task
  - apiMailbox - IPC handle for the send task to write to the API
  - sendMutex – A mutex to coordinate access to the send task mailbox by the receive and API tasks.
  - receiveSemaphore – A semaphore which is in a state of signaled when publications are available to receive.

- **Receive task** - MQIsdp_ReceiveTask(MQISDPTI *pTaskInfo);
  This takes a MQISDPTI structure (MQISDP Task Info) as a parameter, populated as follows:
  - sendMailbox - IPC handle for the receive task write to the send task.
  - receiveMailbox - IPC handle for the receive task to read from.
  - apiMailbox - Not required – leave undefined.
  - sendMutex – A mutex to coordinate access to the send task mailbox by the receive and API tasks.

- receiveSemaphore – Not required – leave undefined.

- **API task**
  The following parameters are passed into MQIsdp_connect:
  - sendMailbox - IPC handle for the API task write to the send task.
  - apiMailbox - IPC handle for the API task to read from.
  - sendMutex – A mutex to coordinate access to the send task mailbox by the receive and API tasks.
  - receiveSemaphore – A semaphore used by MQIsdp_receivePub in order to wait to receive publications.

Chapter 6. Sample Applications

Two sample applications called publish.c and subscribe.c are supplied for Windows and UNIX platforms. They are compiled by the supplied makefile. Both applications start the send and receive tasks as threads in a process, if the code is compiled to run in multiple threads (as it is by default). Both programs also use the file system implementation of the persistence interface.

publish

Publish takes a URL like string as a parameter which contains all the information required to publish a message on a particular topic. 'publish –h' will give usage. The command line is as follows:

publish wmqtt://clientId@broker:port/topic?qos=[0|1|2]&retain=[y|n]&debug=[0|1]
<LWT=topic?qos=[0|1|2]&retain=[y|n]&data=lwtdata> <data>

wmqtt:// - The fixed identifier for the protocol. Must be specified as is.
clientId - A unique identifier up to 23 characters in length
broker - The hostname or dotted decimal IP address of the broker.
port - The IP port of the broker
topic - The topic on which to publish the data. This maybe hierarchical with each hierarchy being delimited by '/'.
qos - Optional parameter. The Quality of Service at which to deliver the publication – 0, 1 or 2
retain - Optional parameter. Should the publication be retained by the broker – y(yes) or n(no).
debug - Optional parameter. Should debug messages be displayed?

Last Will and Testament. An LW&T message may be specified so that it the publisher application terminates unexpectedly then the broker will publish a message on its behalf to indicate that it has ended unexpectedly.

Parameter fields are as described above, plus:
lwtdata - The data to publish in the LW&T message.

data - The data to publish in the message. If no data is specified then the program enters interactive mode allowing messages to be typed into the program. Each line of data entered will cause a message to be published. Pressing return with no data will end interactive mode.

Example usage:

publish "wmqtt://pubId@localhost:1883/my/topic/space?qos=1&retain=n"
"LWT=dead/topic?qos=1&retain=n&data=Publisher_terminated"

This will enter interactive mode and publish all messages to topic my/topic/space, with a LW&T messages of "Publisher_terminated" being sent to topic dead/topic if publish terminates unexpectedly.

Double quotes are required around the parameters to stop the shell interpreting the special characters.

subscribe

Subscribe takes a URL like string as a parameter which contains all the information required to publish a message on a particular topic. 'subscribe –h' will give usage. The command line is as follows:

subscribe wmqtt://clientId@broker:port/topic?qos=[0|1|2]&timeout=secs&debug=[0|1]
<LWT=topic?qos=[0|1|2]&retain=[y|n]&data=lwtdata> <data>

wmqtt:// - The fixed identifier for the protocol. Must be specified as is.
clientId - A unique identifier up to 23 characters in length
broker - The hostname or dotted decimal IP address of the broker.

port - The IP port of the broker
topic - The topic to subscribe to for data. This maybe hierarchical with each hierarchy being delimited by '/'.
qos - Optional parameter. The maximum Quality of Service at which this subscriber can receive data – 0, 1 or 2
timeout - Optional parameter. How long should the program wait for data?. Default is forever.
debug - Optional parameter. Should debug messages be displayed?

Last Will and Testament. An LW&T message may be specified so that it the subscriber application terminates unexpectedly then the broker will publish a message on its behalf to indicate that it has ended unexpectedly.

Parameter fields are as described above, plus:
lwtdata - The data to publish in the LW&T message.

data - Data to match in received messages. If a publication is received with data matching that specified then the application will return, otherwise it will carry on waiting for more publications. The special value of '*' will match the first message received. If no data is specified then the subscriber will receive all messages until the timeout expires.

Example usage:

subscribe "wmqtt://subId@localhost:1883/my/topic/space?qos=1&timeout=60" "LWT=dead/topic?qos=1&retain=n&data=Subscriber_terminated" "*"

This will subscribe to topic my/topic/space, with a LW&T messages of "Subscriber_terminated" being sent to topic dead/topic if publish terminates unexpectedly. The subscriber will return upon the first message being received.

Double quotes are required around the parameters to stop the shell interpreting the special characters.