MQIsdp protocol C language implementation Version 1.0

21 February, 2003

SupportPac author
Ian Harwood

ian_harwood@uk.ibm.com

Property of IBM

Take Note!

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

First Edition, February 2003

This edition applies to Version 1.0 of S*upportPac title* 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
- MQSeries Integrator
- MQSI

The following terms are trademarks of other companies:

- Windows, Microsoft

Summary of Amendments

Date	Changes
21 February 2003	Initial release

Preface

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

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*

Chapter 1. C language MQIsdp API and the programming model

The MQIsdp protocol is built into a shared library on the WIN32 and UNIX platforms (mqisdp.dll and libmqisdp.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

Programming model

The MQIsdp 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 MQIsdp 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 MQIsdp broker. The protocol is in a state of CONNECTING.

MQIsdp_status returns the status of the connection between the device and the MQIsdp 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).

Chapter 2. MQIsdp 'C' language API

Connect

int MQIsdp_connect(MQISDPCH *pHconn ,
CONN_PARMS * pCp);

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

Returns:

- Return code:
  MQISDP_OK
  MQISDP_NO_WILL_TOPIC
  MQISDP_ALREADY_CONNECTED
  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
Version	Long	Structure version Set to value MQISDP_VERSION_1
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	Number of times to retry a failed operation – attempts to connect and attempts to deliver a message.
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 MQIsdp server does not receive any data within this time limit it will assume the application has terminated.
apiMailbox	MBH	Only required if running the MQIsdp protocol in multiple tasks. See the section entitled 'Single vs. Multi task solution' for more information.
sendMailbox	MBH	
sendMutex	MTH	
receiveSemaphore	MSH	
numServers	long	Number of MQIsdp brokers for which connection information is supplied (in the mqisdpSvr field). The MQIsdp send task will cycle through the list until a successful connection is established. If the send task needs to reconnect it will only attempt to reconnect to the previously used connection.
mqisdpSvr	MQISDP_SVR[1]	An array of numServers structures that contain port number and ip address information.
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

- **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 MQIsdp 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_CONN_HANDLE_ERROR
- pHconn - Connection handle is set to MQISDP_INV_CONN_HANDLE

Publish

```
int MQIsdp_publish( MQISDPCH     hConn,
                    MQISDPMH   *pHmsg,
                    PUB_PARMS *pPp,
                    long             dataLength,
                    char            *pData );
```

Inputs:

- hConn - A valid connection handle
- pHmsg - Address of a new message handle
- pPp - Pointer to a PUB_PARMS structure
- dataLength - The length of the data to be published
- pData - Pointer to the data to be published

Returns:

- Return code:
  MQISDP_OK
  MQISDP_CONN_HANDLE_ERROR
  MQISDP_Q_FULL
  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

PUB_PARMS:

Field	Data Type	Usage
version	long	Structure version

		Set to value MQISDP_VERSION_1
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 MQIsdp broker until another publication is received for the same topic.
topicLength	Long	The length of the topic
Variable length portion of structure		
topic	char[n]	The topic to be associated with the data being 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_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**
version	long	Structure version

		Set to value MQISDP_VERSION_1
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 MQISDP_QOS_1 published at to this application by the MQISDP_QOS_2 broker.

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_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**
version	Long	Structure version Set to value MQISDP_VERSION_1

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 MQIsdp client and the MQIsdp 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 MQIsdp protocol successfully connected.
- If the status is MQISDP_DISCONNECTED
  - If pInfoCode is MQISDP_KEEP_ALIVE_TIMEOUT
    then pInfoString contains the time that the MQIsdp server last responded.
  - If pInfoCode is MQISDP_PROTOCOL_VERSION_ERROR
    then the MQIsdp broker cannot support the version of the MQIsdp protocol specified.
  - If pInfoCode is MQISDP_CLIENT_ID_ERROR
    then the MQIsdp broker rejected the client ID for some reason.
  - If pInfoCode is MQISDP_BROKER_UNAVAILABLE
    then the MQIsdp broker rejected the connection because the broker is busy.
  - If pInfoCode is MQISDP_SOCKET_CLOSED
    then the MQIsdp 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 MQIsdp 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 MQIsdp 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_DISCARDED

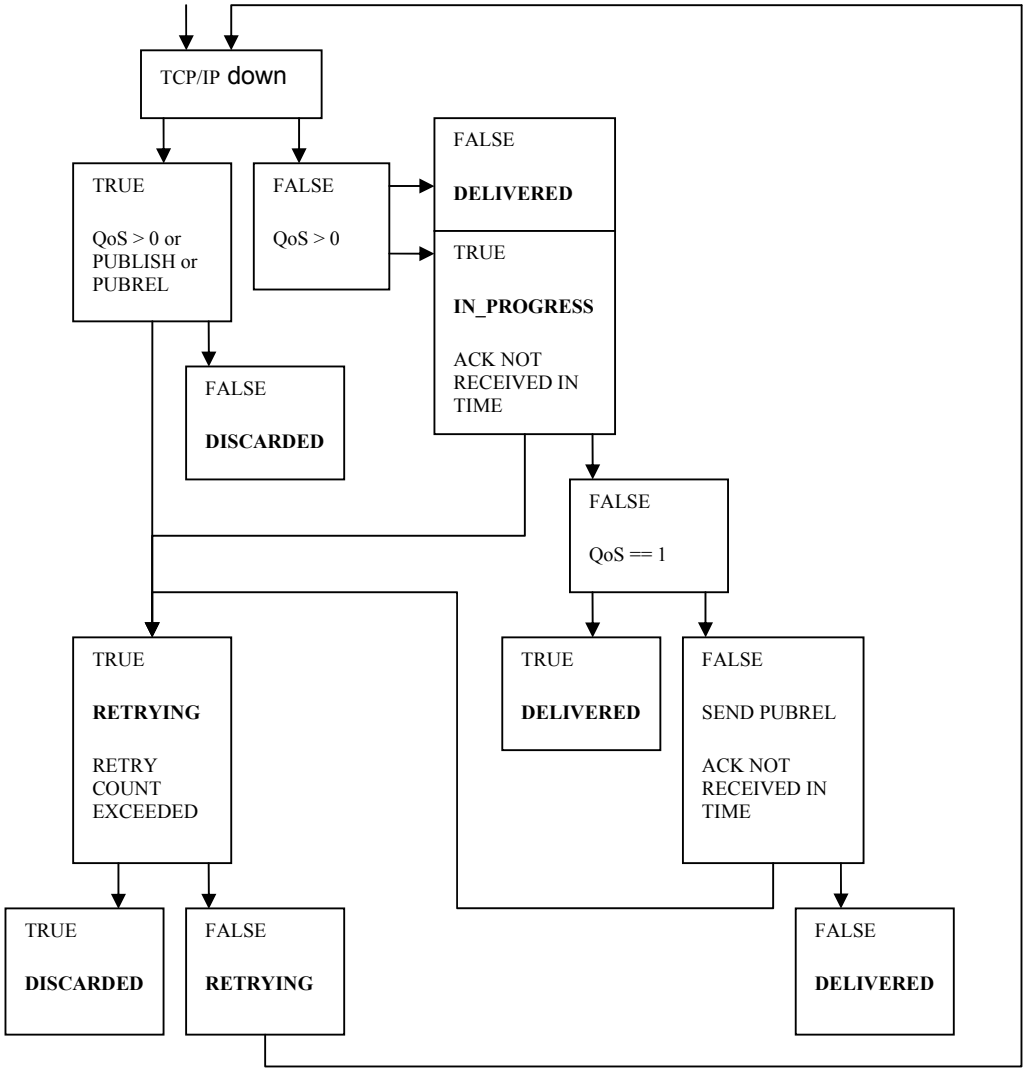
MQISDP_DELIVERED or MQISDP_DISCARDED are the final states than a message can get into. A message is discarded once all retries (as specified when connecting) have been exhausted. A message is delivered once all the Quality of Service MQIsdp protocol flows are complete.

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 MQIsdp message type determine how a message is handled. An understanding of the MQIsdp 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 MQIsdp 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.

Return Codes

Return Code	Value	Explanation
MQISDP_OK	0	Success
MQISDP_PROTOCOL_VERSION_ERROR	1001	The MQIsdp broker does not support this version of the MQIsdp protocol
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_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 MQIsdp 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 MQIsdp client to establish a connection with the MQIsdp 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 MQIsdp broker refused the connection attempt because of a problem with the client identifier.
MQISDP_BROKER_UNAVAILABLE	1020	The MQIsdp 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. Compiling and linking client applications

To compile and link the MQIsdp 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 MQIsdp.dll. To use this DLL an application needs to link with MQIsdp.lib.

Linking on Linux

An application needs to link with libmqisdp.so which contains the MQIsdp protocol.

Chapter 4. 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 MQIsdp 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 MQIsdp 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 CONN_PARMS structure.

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 MQIsdp 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.
- pSendTaskInfo - A pointer to an uninitialised MQISDPTI structure
- pRcvTaskInfo - A pointer to an uninitialised MQISDPTI structure.
- pClientId – A string containing the client identifier that this application will use.

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 5. Sample Applications

Two sample applications called demo.c and democ.c are supplied for Windows 2000 and various 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).

demo

Demo simulates a flowmeter. It randomly publishes numbers in a range for 600 seconds by default. To run the demo application enter: demo –a <broker ip address e.g. 127.0.0.1>

The application will publish data on topic demo/c/flowrate as ASCII characters to a broker on port 1883 of the specified address. demo also subscribes to topic demo/c/control to receive control commands from the democ application (below).

'demo –h' will display more options. By default the application will publish numbers every 5 seconds for 600 seconds. The simulation of a flowmeter will record a flow at changing rates for 120 seconds, followed by no flow for 60 seconds, flipping back and fore between these states for the duration of the application.

democ

Democ is a control application that can change the parameters within which demo operates by publishing commands to topic demo/c/control.

Things that can be changed are the rate of flowrate change, the publication rate of demo, and the max and min bounds in which the flowrate is confined.

'democ –h' will display the exact options.

---------- End of Document ----------