WebSphere MQ Telemetry Transport
C language implementation
Porting Guide
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

Trademarks and service marks

Summary of Amendments

Date	**Changes**
21 February 2003	Initial release
6 June 2003	Version 1.1 release
30 Novemeber 2003	Version 1.2 release • Modified shared library name 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.

This document provides instructions on porting the C implementation to other operating systems.

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. Introduction

Firstly decide whether you want to implement the code so that it runs in a single thread of execution, or in multiple threads of execution.

As a first step it is recommended to compile the code to run in a single thread of execution, as this is a relatively straight forward thing to do before moving on to running the code in multiple threads of execution.

1. To compile the code to run in a single thread of execution define MSP_SINGLE_THREAD when compiling the source code. This requires a C library and a TCP/IP library in order to compile. See Platform Specifics and TCP/IP stacks supported for more information about this.
2. To compile the code to run in multiple threads of execution **do not** define MSP_SINGLE_THREAD when compiling. This requires a C library, a TCP/IP library and some OS specific functionality for doing Inter Process Communication and task coordination. See Inter Process Communication for more information about this.
   There are three threads as follows:
   - 'The send task' – This is the thread that writes to the TCP/IP socket, queues data to sent and received and retries any failed transmissions.
   - 'The receive task' – This is the thread that reads from the TCP/IP socket. It simply passes any received data to the send task.
   - 'The API task' – This is the thread of execution of the user application which uses the WMQTT API.

Chapter 2. Platform Specifics

Platform specific includes and macro definitions should be limited to mspsh.h where possible.

Some useful macros are available, which may be defined, if certain C library functions are not available:

- **MSP_NO_LOCALTIME** - No localtime() function is available. This macro disables some datetime formatting in the mspLog function (mspsh.c)
- **MSP_NO_REALLOC** - No realloc() function is available. mspRealloc (mspsh.c) does a malloc followed by a free instead.
- **MSP_NO_ISPRINT** - No isprint() function is available. This macro disables some formatting in the mspLogHex function (mspsh.c)
- **MSP_NO_TIME** - No time() function is available. This macro causes a time() function prototype to be defined in mspsh.h. A function must be implemented in mspsh.c that conforms to this prototype and returns the number of seconds from an epoch (usually 00:00 January 1st 1970).
- **MSP_NO_USHORT** - No type definition exists for C type u_short (unsigned short). This macro causes a type definition to be generated.
- **MSP_NO_UINT** - No type definition exists for C type u_int (unsigned int). This macro causes a type definition to be generated.
- **MSP_NO_ULONG** - No type definition exists for C type u_long (unsigned long). This macro causes a type definition to be generated.

The following type definitions are required in MQIsdp.h for the following types. By default these types are defined to be int, which is fine if the code is being compiled to run in a single thread, as they are not used.

- MBH – IPC handle (Mailbox HANDLE on Windows 2000, pipe on Unix)
  See Inter Process Communication, Passing data between tasks.
- MTH – Mutex semaphore handle (HANDLE on Windows 2000, int on Unix)
  See Inter Process Communication, Coordinating access to the send task.
- MSH – Resource semaphore handle (HANDLE on Windows 2000, struct on Unix)
  See Inter Process Communication, Enabling MQIsdp_receivePub() to block with a timeout.

Other required macros in mspsh.h are:

- MSP_NULL_MAILBOX - a NULL value for type MBH
- MSP_NULL_MUTEX - a NULL value for type MTH
- MSP_NULL_SEMAPHORE - a NULL value for type MSH
- MSP_IPC_WAIT_FOREVER - a value to indicate that the IPC mechanism should block forever when reading.
- MSP_INVALID_SOCKET - a value for an invalid socket descriptor (usually -1).

Chapter 3. Inter Process Communication (IPC)

The Inter Process Communication mechanism is implemented in the following source files: **mspipc.c, mspsh.h, mspclnt.h, MQIsdp.h**

Passing data between tasks

Implemented in functions: mspInitialiseIPC, mspReadIPC, mspWriteIPC

The Windows 2000 implementation uses mailslots for interprocess communication. The generic UNIX version uses pipes.
A fixed length block of data (structure CB_HEAD, size 16 bytes) is written to a mailslot. Part of this data is a pointer to the data to be passed. The receiving task then reads the data directly from this memory address, so only the CB_HEAD structure is actually passed via the IPC mechanism.

This mechanism relies on tasks being able to address each others memory. This is not a problem for threads in a process or tasks in some DOS based systems. If this is not possible, then the IPC functions to read and write can be adapted to read/write the CB_HEAD structure and the associated data from/to the IPC subsystem.

The C type for the IPC handle is MBH. This is typedef'ed in MQIsdp.h to be a HANDLE on Windows 2000 and an int on UNIX, but can be whatever is appropriate for your platform.

There are three functions to manage the IPC interface:

mspInitialiseIPC – This function is called at task start up for the send and receive tasks. For Windows 2000 it sets a mailslot read timeout value. On UNIX the signal SIGPIPE is masked out.

mspReadIPC – This function accepts data from the IPC interface, copying the data into a buffer supplied by the caller and resizing the buffer if required. Information from the CB_HEAD structure is also passed back.
This function is used by all tasks. The API and receive tasks do blocking reads of the IPC interface, whilst the send task reads with a timeout, as it has other things to do such as retrying sends and managing the TCP/IP connection. For Windows the read timeout is set when the mailslot is initialised. On UNIX select is used prior to every read to wait for data.

mspWriteIPC – This function writes a CB_HEAD structure to the IPC interface. It accepts a data buffer plus various parameters to place in the CB_HEAD structure.

If you need to store extra parameters for your particular IPC interface, then each call also accepts an IPCCB structure (IPC Control Block), with various parameters in, such as a read timeout. Any additional parameters can be added to this structure which is defined in mspsh.h.

Notes: Behaviour when compiling as a single task.

When compiling to run in a single thread a fixed length buffer the size of the CB_HEAD structure is allocated in the MSPCCB (Client Control Block) structure defined in mspclnt.h. mspWriteIPC writes to the buffer and mspReadIPC reads from this buffer. This method keeps the mechanism of passing data between the API and the send task code relatively similar to the multi task solution, even though it is now all running in one thread of execution.
Effectively it is now a mechanism for passing data up and down the stack instead of between tasks.

Coordinating access to the send task

Implemented in functions: mspLockMutex, mspReleaseMutex

The send task (MQIsdp_SendTask(MQISDPTI *) in mspdmn.c) is the main worker task. It manages the TCP/IP connection, queues up and retries any data to be sent to the WMQTT broker and queues up data to be received by the application. The send tasks receives data via IPC whenever an API call is made and whenever the receive task receives data from the network. To keep the implementation as simple as possible, each task can only have one request at a time to process in its mailslot. Consequently the API and receive tasks must take it in turns to write to the send mailslot.

Before attempting to write to the send task the sendMutex is obtained by calling mspLockMutex. Data is then written to the send task and mspReleaseMutex is called when a response has been received for the send task.

The C type for the mutex handle is MTH. This is typedef'ed in MQIsdp.h to be a HANDLE on Windows 2000, and an integer semaphore id on UNIX, but can be whatever is appropriate for your platform.

The behaviour required for the mutex is that it is normally unlocked (available), only being locked for the duration that an application is writing data to the send thread.

Notes: Behaviour when compiling as a single task.

These functions should just return 0 when compiled to run in a single thread of execution as no inter-process coordination is required.

Enabling MQIsdp_receivePub() to block with a timeout

Implemented in functions: mspWaitForSemaphore, mspSignalSemaphore, MQIsdp_receivePub

It is optional whether these functions are implemented or not. If blocking receives are not required then these functions need only to return 0. MQIsdp_receivePub will then return immediately indicating whether publications are available or not.

Blocking in a multi-task environment

In a multitask environment the MQIsdp_receivePub API calls mspWaitForSemaphore with a timeout. If the semaphore times out then MQISDP_NO_PUBS_AVAILABLE is returned, otherwise the API call proceeds to retrieve the publication.
The semaphore is signaled in a number of circumstances:

1. A Quality of Service 0 or 1 publication arrives and the number of publications available to receive changes from 0 to 1.
2. A Quality of Service 2 PUBREL message is received and the number of publications available to receive changes from 0 to 1.
3. A MQIsdp_receivePub call successfully completes and there are more publications available.
4. A MQIsdp_receivePub call fails with MQISDP_DATA_TRUNCATED meaning the current publication has not been received.

When mspWaitForSemaphore completes it should ensure that the semaphore is left in a state that indicates no messages are available, regardless of whether further messages are available or not. The semaphore will be signaled again if appropriate once the current publication has been successfully received.

The C type for the semaphore handle is MSH. This is typedef'ed in MQIsdp.h to be a HANDLE on Windows 2000, and a structure on UNIX, but can be whatever is appropriate for your platform.
On UNIX the MSH_S structure (see MQIsdp.h) contains a pthread mutex, a pthread condition variable for doing timed waits and a char variable that holds the message available state. UNIX semaphores cannot be used, because timed waits cannot be done on UNIX semaphores.

The behaviour required for the semaphore is that it is normally not signaled, only being in a state of signaled when a message is available. The semaphore mechanism needs to allow timed waits to be done.

Blocking in a single task environment

In a single task environment a blocking receive is not so easy to implement. The API cannot simply block on the TCP/IP socket waiting for data to arrive because there are other things to do such as retrying messages, managing the TCP/IP connection and keeping the WMQTT connection alive.

A loop is implemented in MQIsdp_receivePub, which calls the send task code. The send task code blocks on the TCP/IP socket for up to 50% of the keepalive interval. It then checks for any retries required and the state of the WMQTT connection before returning to the API stating how long it has waited. The API then deducts this interval from the wait time remaining and calls the send task code again until the wait time remaining has reached 0.

Notes: Behaviour when compiling as a single task.

mspWaitForSemaphore and mspSignalSemaphore should just return 0 when compiled to run in a single thread of execution as blocking receives using semaphores cannot be done.

Chapter 4. TCP/IP stacks supported

Implemented in source files: msptcp.c mspdmn.h

The code was originally written to support the BSD (Berkeley Sockets) socket style interface. This is the TCP/IP interface that will be used by default. Other TCP/IP socket style interfaces have since been added:

- Softworks Group (a unit of NetSilicon) Fusion™ TCP/IP
  Define MSP_FUSION_SOCKETS in mspsh.h when compiling.
- KADAK KwikNet® TCP/IP
  Define MSP_KN_SOCKETS in mspsh.h when compiling.

If another TCP/IP interface is to be supported then create a macro name that can be defined at compile time to enable the code to use that interface. Wrapper functions have been created for the socket API calls used, which handle the differences between the various TCP/IP implementations:

- msp_close
- msp_connect
- msp_inet_addr
- msp_recv
- msp_select
- msp_send
- msp_socket

Chapter 5. Memory Management

Implemented in source files: mspsh.c mspsh.h
Implemented in functions: mspMalloc, mspFree, mspRealloc

Wrapper functions are supplied for doing memory management as named above. These functions currently do not impose any memory limits on the protocol.

Reasons for increasing memory usage

There are four possible reasons as to why memory usage might increase:

1. A high publication rate to the broker.
   If the client application publishes data faster than the broker can process the data then things will inevitably backup and memory will be consumed queuing data up to send.
2. A high publication rate from the broker.
   If the broker publishes data at a rate higher than the application can process the data then the protocol will queue data locally on the device ready to be received.
3. The application calls MQIsdp_receivePub infrequently.
   If the application does not receive data frequently enough then the protocol consumes more memory queuing up publications.
4. If a TCP/IP error occurs then the protocol queues up data whilst it retries to establish the TCP/IP connection.

Under normal operation the protocol should use less than 2K of heap with a message size of less than 100 hundred bytes.

Memory usage controls

Current memory controls in place are defined in mspsh.h:

- Maximum amount of data that can be queued for sending - MSP_DEFAULT_MAX_OUTQ_SZ - 32Kb.
  This size includes all memory required to queue a message, not just the data length that is being sent. The application will get returned MQISDP_Q_FULL when this limit is reached.
- Maximum amount of data that can be queued for receiving - MSP_DEFAULT_MAX_INQ_SZ - 32Kb.
  This size includes all memory required to queue a message, not just the data length that is being received. If this queue fills up the following will happen:
  - QoS 0 messages will be discarded, as they cannot be received
  - QoS1 messages will not have the PUBACK sent
  - QoS2 messages will not have the PUBREC sent
  - The WMQTT broker will at a later time resend any QoS1 or QoS2 message for which it doesn’t receive an acknowledgement.
- Initial size of the IPC memory buffer for each task – MSP_DEFAULT_IPC_BUFFER_SZ - 128 bytes. This will dynamically grow as required.
- Number of keys in the internal hash tables. MSP_DEFAULT_NUM_HASH_KEYS - 16.
  There are two hash tables, one for sending and one for receiving. 16 keys results in an empty hash table of size 68 bytes and each additional key costs 4 bytes.
  A hash table entry is created for each QoS 1 and 2 message sent and each QoS 2 message received. Each entry is 16 bytes plus the size of the WMQTT message in size.

If you need more control of the memory usage of the protocol then the mspMalloc, mspFree and mspRealloc functions can be modified to suit your needs. The memory management calls take a parameter of a pointer to the MSPCMN structure (which is defined in mspsh.h). Any additional information required can be stored in this structure

Debugging memory usage

Macro MSP_DEBUG_MEM is defined in mspsh.h.

Setting its value to 0 will result in no memory debugging being done.
Setting its value to 1 will result in a summary being display for each task, recording:

1. The number of calls to mspMalloc.
2. The number of calls to mspFree
3. The amount of memory currently allocated.
4. The maximum amount of memory in use during the life of the task.

Setting its value to 2 will result in the summary as described above, plus each memory address allocated and freed being recorded. Obviously this produces a lot of output, but can be useful.

Chapter 6. Enabling debug messages

Various levels of debugging can be enabled by setting up the debug level correctly for each component.

Debug Levels are:

LOGERROR	: Log error conditions only
LOGNORMAL	: Log normal operation messages.
LOGIPC	: Log Inter Process Communication flows
LOGTCPIP	: Log TCP/IP data sent and received
LOGSCADA	: Log WMQTT protocol flows
LOGDEBUG	: Log any debug messages

The bitwise OR operation can be used to enable logging of multiple areas of the code. E.g. LOGTCPIP | LOGSCADA will display the protocol flows and associated data being flowed to/from the broker.

Debug messages may be enabled by setting the logLevel in the MQISDPTI structure prior to starting the send task (MQIsdp_SendTask), the receive task (MQIsdp_ReceiveTask) and calling MQIsdp_connect.
The debug level is held in the mspLogOpions field of the MSPCMN structure as follows:

API: Function: MQIsdp_connect, File: mspclnt.c
Set the value of pConnHandle->comParms.mspLogOptions as required.

SEND TASK: Function: mspInitialise, File: msputils.c
Set the value of pHconn->comParms.mspLogOptions as required.

RECEIVE TASK: Function MQIsdp_ReceiveTask, File: mspdmn.c
Set the value of rcvHconn.comParms.mspLogOptions as required.

Chapter 7. Compiling the code and sample applications

All the code except publish.c, subscribe.c and mspfp.c is compiled into a single shared library called wmqtt.dll on Windows and libwmqtt.so on Linux.

To assist in compiling the full source code on Windows 2000 and Linux for the WMQTT protocol there is a makefile which works with GNU make. The makefile can also be used to compile the code for AIX, HP-UX and SUN Solaris.

Pre requisite software:

GNU make is available from http://www.gnu.org/software/make/make.html. Version 3.77 or later is required.
On Windows the Microsoft Visual Studio compiler is required (Version 6.0 or later)
On Linux the GNU cc compiler is required.

When the code is unzipped the source code is in the root directory of the zip file and there are four subdirectories – bin_win, bin_linux_i386, build and doc.

cd into bin_win, bin_linux or a new directory
run 'make –f ../build/Makefile' to compile the code to run in multiple threads.

run 'make –f ../build/Makefile BT=SINGLE' to compile the code to run in a single thread. The BT=SINGLE flag has the effect of defining MSP_SINGLE_THREAD when compiling the code.

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