Message Queuing Service Support in Windows CE

Abstract

Microsoft® Windows® CE is a scalable and compact operating system based on a subset of the WIN32 API. Its multithreaded, fully preemptive nature is designed specifically for small hardware devices. Its modular design enables developers of embedded systems and applications to customize it for a wide range of products, such as consumer electronic devices, specialized industrial controllers, and embedded communications devices.

This paper describes message-queuing-service (MSMQ) support in Windows CE, introduced in Windows CE version 2.12. It discusses the programming model and architecture, MSMQ administration, and MSMQ functions; provides code samples; and identifies differences you will encounter between the Windows CE–based version and the desktop-based version of MSMQ.

System Requirements

Following are system requirements for deploying MSMQ on a Windows CE–based device:

- MinComm is the smallest configuration on which MSMQ can be deployed.
- 150–200K ROM is required to maintain necessary DLLs and the MSMQAdm administration utility.
- The computer receiving messages must have MSMQ installed and must be running the Windows CE, Windows NT® Server, Windows NT® Workstation, or Windows 9x operating system.
- Message Queuing applications can be developed using the C APIs.

Programming Model

The key to building a distributed application that runs on a Windows CE–based device is to provide a way for the various parts of the application to communicate with one other. One option available to Windows CE developers is to use Microsoft® Message Queuing Service (MSMQ), a service that implements asynchronous communications. MSMQ provides guaranteed message delivery, efficient routing, and priority-based messaging.

The Windows CE–based implementation of MSMQ was designed to retain the essential features of the desktop version of MSMQ while streamlining functionality to minimize space requirements. The implementation is geared toward a small number of simultaneous connections but uses short code paths. Other advantages include the following:

- Use of names internally, rather than as IP addresses, supports roaming.
- NIC (network interface card) tracking allows the service to restart immediately after reconnection.
- Transparent, salvageable storage enhances efficiency.

- Routing for nonprivate queues is simple to configure and use and provides powerful routing functionality.
- Scripting-capable console-based administration utility allows telnetting.

Message Queue Architecture

MSMQ technology enables applications running at different times to communicate across networks, computers, and other devices that may be temporarily offline. To accomplish this, the communicating applications send requests as messages to various message queues. Each receiving application reads the message stored in the queue when the connection is re-established. The following diagram illustrates schematically how a queue can hold the messages used by both sending and receiving applications.

Public Queues

In contrast to the desktop version of MSMQ, the Windows CE version of MSMQ does not require connection to a domain controller. Instead, it uses a peer-to-peer configuration that does not use a directory-service interface such as Microsoft Active Directory™. Because information about public queues is stored in Active Directory, you cannot create public queues in Windows CE.

Additionally, Windows CE does not use the direct format name model for naming queues, because a direct-only format name typically requires a direct TCP connection between two computers. Moreover, the desktop application would need to be aware of the Windows CE version of MSMQ and would always have to specify the response queues used for reply messages in direct-only format. For these reasons, the direct-only architecture is not an optimal solution for queuing messages in Windows CE.

Windows CE MSMQ can, however, send messages to public queues located on a desktop computer or server through the use of an *OutFRS queue*. An OutFRS queue is an outgoing queue that points to a Falcon Routing Server (FRS). The target computer must be a message queue server capable of routing messages; it cannot be another Windows CE–based device or a desktop-independent client. When a Windows CE–based device connects to an OutFRS server, messages from an outgoing OutFRS queue are moved to the server. This server is then responsible for the correct routing of the messages.

An OutFRS queue is specified as a direct format name in an **OutFRSQueue** registry value. However, the queue itself does not have to exist on the server, because the destination queue is included as part of the message address and is specified in the **PROPID_M_DEST_QUEUE** message property, which supports messages sent to **OutFRSQueue**. Additionally, all administrative responses generated by MSMQ, such as acknowledgements (ACKs) and negative acknowledgements (NACKs), are channeled automatically through an OutFRS queue if a non-direct queue name is specified as an administrative queue for a message.

Queuing Scenarios

Based on the architecture previously described, the following queuing scenarios are possible:

- An intermittently connected Windows CE–based device (A) sends a message to another intermittently connected Windows CE–based device (B) by means of a routing server (C). Messages are sent from A to B through an OutFRS queue pointing to C. When A connects to C, messages are moved to C and are held until B connects to C, after which they are delivered to B.
- A Windows CE–based device is running on a secure subnet that is protected by a firewall from a destination computer. A routing server bridges the firewall and delivers messages to a remote computer.
- A desktop application typically specifies its own queue in a public format as a target of an administrative acknowledgement. A Windows CE–based device automatically forwards ACKs or NACKs to this queue through an OutFRS queue, precluding the need to rewrite the application or create a special-case Windows CE queue.

Transaction Support

The Windows CE–based MSMQ server programming model supports one type of transaction: a single message transaction. The single message transaction orders messages and guarantees only a single delivery of each message. To guarantee single, in-order delivery, you must create the queue as transactional, and messages must be sent to the queue using the **MQ_SINGLE_MESSAGE** transaction constant.

MSMQ Features Not Supported in Windows CE

The Windows CE–based message queuing service does not support the following features:

- Remote Procedure Calls (RPCs)
- Encryption
- Access Control List (ACL)–based security
- MQMail
- Dependent client functionality

Additionally, Windows CE supports only TCP/IP network stacks.

Administration

Windows CE implements an independent client that is not part of any domain and stores all relevant administration information on the device itself. Administration functions are carried out locally through MSMQAdm, a utility that configures MSMQ. MSMQAdm is script-driven and console-based, enabling developers to automate routine tasks. Tasks administered through MSMQAdm include the following:

- Browsing local queues
- Purging messages
- Deleting individual messages
- Stopping and starting MSMQ
- Connecting and disconnecting from the network

Configuring MSMQ

The entire persistent state of MSMQ is a combination of information in the MSMQ registry key and the MSMQ base directory. The base registry key is **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSMQ\SimpleClient**. The following table provides the named values for the base-registry MSMQ configuration parameters.

Value	Description
Port	The machine port. This port is always set to 1,801 by the registration utility. Do not change the port setting.
PingPort	The machine ping port. This ping port is always set to 3,527 by the registration utility. Do not change the ping-port setting.
DefaultQuota	The queue quota for message storage by a remote queue. If this value does not exist, outgoing queues are unrestricted.
QueueManagerGUID	The globally unique identifier (GUID) of the Windows CE queue manager, used for session protocols between Windows CE and Windows NT implementations.
OutFRSQueue	The outgoing FRS queue format name. This name must be DIRECT. If any other name is used, it will be impossible to configure the outgoing FRS queue.
DebugQueue	The queue used for routing debugging.
BaseDir	The directory used for queue information and message storage.
HostName	The machine host name. If this is not set, information is queried dynamically from the network.
MessageID	The current message identifier, accurate within 1,000 messages. This identifier is set and used by the MSMQ client and should not be changed by a user.
CEStartAtBoot	Windows CE–specific entry. If this value is a 0 DWORD entry, MSMQ will not start at boot time but can be started using the MSMQ administration utility.

The base directory contains both local and outgoing queue and message data as well as MSMQ internal queues. One file is contained in each queue. At startup, MSMQ creates a machine journal queue and dead-letter queue, if they are not already present. These queues have a local queue manager GUID as part of their names. The following table describes file types associated with GUIDs.

Extension	File type
Oq	Outgoing messages that were sent to queues on different machines
iq	Local queues and messages residing in these queues
Jq	Journal counterparts to local queues

File names are prefixed with the machine name and “public” token for public queues. The exception is the naming convention for local queues, which are prefixed with the “$localhost$” token and cannot be public.

Configuration information should be present for MSMQ to start at boot time. If the registry key is not configured, MSMQ starts in a suspended state. After configuration is complete, you can start MSMQ using the MSMQAdm utility.

Note

You must configure MSMQ after you set the computer name and current time. Current time is used for the generation of the GUID. If time is not set properly, more than one machine can end up with the same GUID. For this reason, configuration does not proceed automatically during the first cold boot but instead is left to the user to accomplish.

To use MSMQAdm, run it with its command as an argument. The following example shows how to run MSMQAdm in both automatic and manual modes.
Msmqadm <command>
Msmqadm man

MSMQAdm displays a prompt so that you can type commands interactively. To obtain a list of commands, type the following command:
Msmqadm help

The following table identifies some additional commands.

Command	Description
Man <filename>	Executes a script from a file
Exit	Exits manual mode
Enum queues	Displays a list of queues present on a device

Enum messages <queue number>	Enumerates messages in a specified queue
Delete queue <queue number>	Deletes a queue
Delete message <message number>	Deletes a message
Purge queue <queue number>	Purges a queue and deletes all messages
Start	Starts the MSMQ service
Stop	Stops the MSMQ service
Register {GUID}	Creates the MSMQ configuration in the registry. The optional GUID specifies the GUID of the queue manager
Status	Provides the status of the queue manager
Net connect	Forces all queues to connect to the network
Net disconnect	Abandons all network connections
Tidy	Forces the system to perform an internal data compaction

The following three examples show how to set up and start MSMQ on the new device. Enter the commands from the command prompt.

```
\> msmqadm register
\> msmqadm start
```

or

```
\> msmqadm man
>register
>start
>exit
\>
```

or

```
\> echo register > start.txt
\> echo start >> start.txt
\> echo exit >> start.txt
\> msmqadm man start.txt
```

MSMQ Functions in Windows CE

The following table shows the level of support provided by Windows CE for MSMQ functions.

MSMQ Function	Windows CE Implementation
MQBeginTransaction	Not supported. Returns **MQ_ERROR_SERVICE_NOT_AVAILABLE**.
MQCloseCursor	Fully supported.
MQCloseQueue	Fully supported.
MQCreateCursor	Fully supported.
MQCreateQueue	Can create only local private queue. Some properties specified with the *pQueueProps* parameter have limited support on Windows CE. For more information, see the Windows CE SDK. By default, the M**QCreateQueue** function creates queues that have no journal. For these queues, you cannot activate message journaling using **MQSetQueueProperties**. To create a queue that has a journal, use the **PROPID_Q_JOURNAL** property set to **MQ_JOURNAL**. This creates a queue with a journal and turns journaling on. To turn journaling off, call **MQSetQueueProperties**.
MQDeleteQueue	Can delete only local queue using DIRECT format name.
MQFreeMemory	Fully supported.
MQFreeSecurityContext	Not supported.
MQGetMachineProperties	Can be called only for local machine. The only property supported is P**ROPID_QM_MACHINE_ID**.
MQGetQueueProperties	Can be called only on local machine using DIRECT format name. The **PROPID_Q_INSTANCE** property is not supported. Returns **MQ_INFORMATION_PROPERTY_IGNORED** in status field.
MQGetQueueSecurity	Not supported. Returns **MQ_ERROR_SERVICE_NOT_AVAILABLE**.
MQGetSecurityContext	Not supported. Returns **MQ_ERROR_SERVICE_NOT_AVAILABLE**.
MQHandleToFormatName	Returns DIRECT format name.

MQInstanceToFormatName	Not supported. Returns **MQ_ERROR_SERVICE_NOT_AVAILABLE**.
MQLocateBegin	Not supported. Returns **MQ_ERROR_SERVICE_NOT_AVAILABLE**.
MQLocateEnd	Not supported. Returns **MQ_ERROR_SERVICE_NOT_AVAILABLE**.
MQLocateNext	Not supported. Returns **MQ_ERROR_SERVICE_NOT_AVAILABLE**.
MQOpenQueue	Can open queues only by means of PRIVATE DIRECT format name. Never fails to open outgoing queue if format name is correct and disk space is sufficient.
MQPathNameToFormatName	Returns PRIVATE DIRECT format name.
MQReceiveMessage	The *pTransaction* parameter is not supported. Set it to NULL. If the following encryption properties are specified, they return **MQ_ERROR_COMPUTER_DOES_NOT_SUPPORT_ENCRYPTION**: **PROPID_M_HASH_ALG** **PROPID_M_ENCRYPTION_ALG** **PROPID_M_PROV_TYPE** **PROPID_M_SECURITY_CONTEXT** Other encryption properties return NULL data.
MQSendMessage	Only single message transactions are supported. Some properties are not supported. For more information, see the Windows CE SDK.
MQSetQueueProperties	Can set properties only on local queues using DIRECT format name. **PROPID_Q_AUTHENTICATE** can be only **MQ_AUTHENTICATE_NONE**. **PROPID_Q_JOURNAL** is supported only for queues created with the journal property. For more information, see the Windows CE SDK.
MQSetQueueSecurity	Not supported. Returns

	MQ_ERROR_SERVICE_NOT_AVAILABLE.

MQCreateQueue Implementation

The Windows CE implementation of the **MQCreateQueue** function differs from its implementation on Windows-based desktop platforms in the following ways:

- **MQCreateQueue** can create only a local private queue.
- The *pSecurityDescriptor* parameter must be NULL.

The following table shows properties specified with the *pQueueProps* parameter that have limited support on Windows CE.

Property	Windows CE Support
PROPID_Q_AUTHENTICATE	Only **MQ_AUTHENTICATE_NONE** is enabled.
PROPID_Q_BASEPRIORITY	Accepted but not effective because there is no way to query it from outside.
PROPID_Q_PATHNAME	Only private path names are enabled.
PROPID_Q_PRIV_LEVEL	Only **MQ_PRIV_LEVEL_NONE** is enabled.
PROPID_Q_TRANSACTION	Only **MQ_TRANSACTIONAL_NONE** is enabled.
PROPID_Q_TYPE	Accepted but useless since there is no way to query it from outside.

By default, the **MQCreateQueue** function creates queues that have no journal. For these queues, you cannot activate message journaling using **MQSetQueueProperties**. To create a queue that has a journal, you must use the **PROPID_Q_JOURNAL** property set to **MQ_JOURNAL**. This creates a queue with a journal and turns journaling on. To turn journaling off, call **MQSetQueueProperties**.

MQSendMessage Implementation

The Windows CE implementation of the **MQSendMessage** function differs from its implementation on Windows-based desktop platforms in the following ways:

- The *pTransaction* parameter must be **NULL** or **MQ_SINGLE_MESSAGE**.
- The **PROPID_M_AUTH_LEVEL** property can be only **MQMSG_AUTH_LEVEL_NONE**.
- When a message is sent to an OutFRS queue, **PROPID_M_DEST_QUEUE** is supported providing it uses the format name of an actual target queue.

- The following properties are not supported and, if specified, return **MQ_ERROR_PROPERTY**:

PROPID_M_XACT_STATUS_QUEUE	**PROPID_M_PROV_TYPE**
PROPID_M_XACT_STATUS_QUEUE_LEN	**PROPID_M_PROV_NAME_LEN**
PROPID_M_SIGNATURE	**PROPID_M_PROV_NAME**
PROPID_M_SIGNATURE_LEN	**PROPID_M_PRIV_LEVEL**
PROPID_M_SENDERID_LEN	**PROPID_M_HASH_ALG**
PROPID_M_SENDERID_TYPE	**PROPID_M_ENCRYPTION_ALG**
PROPID_M_SENDERID	**PROPID_M_DEST_SYMM_KEY**
PROPID_M_SENDER_CERT_LEN	**PROPID_M_DEST_SYMM_KEY_LEN**
PROPID_M_SENDER_CERT	**PROPID_M_CONNECTOR_TYPE**
PROPID_M_SECURITY_CONTEXT	**PROPID_M_AUTHENTICATED**

Sample Code

The following sample shows how to create a queue, send a message, receive a message synchronously, and receive a message asynchronously with events through the use of **CMQWrap**. The **CMQWrap** class hides the complex data structures of MSMQ so developers can prototype MSMQ applications quickly.

This sample has two parts:

- MSMQTest.cpp implements a dialog-based application that is a simple client for MSMQ. It can create a queue, send messages, and receive messages. For receiving messages, it supports Synchronous Receive, Asynchronous Receive with callbacks, and Asynchronous Receive with events.
- The **CMQWrap** class is a custom class that hides the complex data structures of MSMQ to allow quick prototyping of MSMQ applications. The use of the **CMQWrap** class is demonstrated in MSMQTest.cpp.

Compiling the Sample Application

MSMQTest_NT.exe is a Win32 application that will run on Windows NT, Windows 9x, and Windows 2000 platforms on which an MSMQ client has been installed. Because WinCE supports a large number of processor types, binaries are not provided. You can compile the binaries yourself using the included source code. MSMQTest.zip contains all the source code for the sample application.

Following is source code for the sample application.

You can compile and run the code on a Windows NT, Windows 9x, or Windows 2000–based platform using the Microsoft Visual C++® development system or another C++ compiler with the Win32 Platform Software Development Kit (SDK). You must install and configure the MSMQ service before running the sample application.

Using the Sample Application

The following illustration shows the sample application interface.

Following are details on how to use the sample application.

1. Click **Create** to create the queue shown in the **Machine\Queue** edit field.
2. Click **Send** to send the message in the **Send Text** edit field to the queue shown in the **Machine\Queue** edit field.

   After each operation, the return value from the message queue API will be displayed in the **Return Code** edit field.

1. Click **Receive** to receive a message synchronously from the queue shown in the **Machine\Queue** edit field. The call will time out if there are no messages in the queue. If a message is received, it will be displayed in the multiline edit field at the bottom of the dialog box.
2. To receive messages asynchronously with a thread/event mechanism from the queue shown in the **Machine\Queue** edit field, select the **Async Receive with Thread/Event** radio button and click **Start Async Receive**. When you click **Send** the message will be automatically dequeued and displayed in the multiline edit field at the bottom of the dialog box.
3. To receive a message asynchronously with a callback, repeat step 5 after selecting the **Asynch Receive with Callback** radio button.

You also can start two copies of the application and send messages between them.

Code Examples

Following are code examples detailing the use of the **CMQWrap** class to create a queue, send a message, receive a message synchronously, and receive a message asynchronously with callback.

To create a queue

The following code example shows how to create a queue using the **CMQWrap** class.

```
CMQWrap q;
q.Create(MQ_SEND_ACCESS, L".\\private$\\myqueue", L"Created by MSMQTest");
```

The **CMQWrap::Create** function creates and opens a queue in two steps. First, it uses the **MQCreateQueue** API to create a data structure containing two properties

for the new queue: Name (**PROPID_Q_PATHNAME**) and Label (**PROPID_Q_LABEL**). After creating a queue, it opens the queue by calling the **CMQWrap::Open** function. **MQWrap::Open** uses **MQOpenQueue** to open the queue that was just created.

To send a message

The following code example shows how to send a message using the **CMQWrap** class.

```
CMQWrap q;
q.Open(MQ_SEND_ACCESS, L".\\private$\MyQueue");
q.SendMessage(L"This is the message body", L"MSMQTest Message");
```

CMQWrap::SendMessage works much like the **CMQWrap::Create** API. It assembles an array of properties for the queue message and then uses the **MQSendMessage** API to queue the property array on a queue. The **MQWrap** class overloads the **SendMessage** function to show how a UNICODE string can be mapped to a queue message. Any arbitrary data structure can be mapped in a similar fashion to a MSMQ message body because the final form for the message body is an array of bytes. **CMQWrap::SendMessage** passes three properties to the **MQSendMessage** API: Message Label **(PROPID_M_LABEL**), Message Body Type (**PROPID_M_BODY_TYPE**) and Message Body (**PROPID_M_BODY**).

To receive a message synchronously

The following code example shows how to receive a message synchronously using the **CMQWrap** class.

```
WCHAR wszBuffer[255];
CMQWrap q;
q.Open(MQ_RECEIVE_ACCESS, L".\\private$\\MyQueue");
DWORD dw = sizeof(wszBuffer);
q.ReceiveMessage(wszBuffer, &dw, INFINITE);
```

CMQWrap::ReceiveMessage receives a message in two steps. First it “peeks” on the queue **(MQ_ACTION_PEEK_CURRENT)** to determine the size of the current message body. After determining the size of the message body, it ensures the buffer supplied by the caller is the correct size and dequeues the message **(MQ_ACTION_RECEIVE)**. Both the peek and the final dequeue operation are performed with the **MQReceiveMessage** API. Note that the action **(MQ_ACTION_RECEIVE)** is the default value for the *dwAction* parameter of the **CMQWrap::ReceiveMessage** function and is therefore not supplied in the preceding code sample. You can choose not to dequeue the message by supplying another action value, such as **(MQ_ACTION_PEEK)**.

To receive a message asynchronously with a callback

1. Define a callback function and global **CMQWrap** variable. For example,

```
// This variable will be used to for asynchronous reads
// via a callback or a thread/event
```

```
CMQWrap g_AsyncQ;
void WINAPI AsyncCallback(CMQWrap* pThis, HRESULT hr, VARTYPE vt,
DWORD dwSize)
{
// Do a receive in here as needed using the CMQWrap queue variable pThis
}
```

2. Start the callback mechanism. For example,

```
g_AsyncQ.Open(MQ_RECEIVE_ACCESS, T2W(wszBuffer, szBuffer));
g_AsyncQ.DoAsyncReceiveWithThread(AsyncCallback); // for using events
g_AsyncQ.DoAsyncReceiveWithCallback(AsyncCallback); // for using callbacks
```

The **MQReceiveMessage** can use two mechanisms to notify the caller asynchronously: a callback function and a Win32 event object. Both mechanisms work on the principle that the **MQReceiveMessage** call completes immediately and the caller is notified asynchronously when a message arrives in a queue. The **CMQWrap** class encapsulates both these mechanisms and passes the notification back to the caller using a **CMQWrap-**defined callback function.

To receive messages asynchronously using Win32 events, the **CMQWrap::DoAsyncReceiveWithThread** function spawns a worker thread that calls **MQReceiveMessage** with a Win32 event object. The worker thread blocks on the event object after the **MQReceiveMessage** call. When a message arrives in the queue, the Win32 event object is signaled by MSMQ and the worker thread is activated and calls the caller's callback function.

Receiving messages using **CMQWrap::DoAsyncReceiveWithCallback** is less complex than using Win32 events. **CMQWrap::DoAsyncReceiveWithCallback** simply calls **MQReceiveMessage** with a private callback provided in the **CMQWrap** class. MSMQ calls this private callback function when a message arrives in the queue. The private callback function forwards the call to the callback function supplied by the caller. In both cases the caller is told that a message is in the queue and given details on the type and size of the message. The caller can dequeue the message in the callback at his or her discretion.

For More Information

For more information on MSMQ concepts, see Application Services on the Windows NT Server home site at http://www.microsoft.com/ntserver/ and the Win32 Platform Software Development Kit.

For information on developing applications for Windows CE, see the Windows CE Platform Software Development Kit.

For papers and articles about Windows CE, see the Microsoft Windows CE Web site at http://www.microsoft.com/windowsce .