API .NET Component

Release 2.3.0

User Manual

This documents contains the installation requirements and procedures for the **MessageMedia** API .NET Component and provides code examples for the use of the component.

Last updated: 11 March 2009

© Copyright 2010 **MessageMedia** Pty Ltd

Contents

1. Product Information 2
2. Audience 3
3. Introduction 3
4. Messaging Platform 3
5. Requirements 5
6. Constraints 5
7. Using the API .NET Component 5
   7.1 Installation 5
   7.2 Setting up your Connection 6
      7.2.1 Configuring Gateway Settings 7
      7.2.2 Configuring Proxy Settings 7
      7.2.3 Configuring Account Settings 8
   7.3 Event Handling 9
   7.4 Sending a Message 11
   7.5 Receiving a Message 13
   7.6 Obtaining Delivery Reports 14
   7.7 Checking your Account Credit 14
8. Contact Us 15

1. Product Information

Product Name **MessageMedia** API .NET Component

Purpose This document lays out the requirements and specifications for the integration of the **MessageMedia** API .NET Component.

Release			
	1.0.1440.19341	15 December 2003	Created
	1.0.1493.23304	2 February 2004	1st Revision
	1.0.1977.21305	31 May 2005	2nd Revision
	1.1.0.0	10 January 2006	1st Release
	1.1.0.2	8 November 2007	2nd Release
	2.0.0	18 August 2008	Pre-release. API Completely rewritten in C# .NET using an event-driven model.
	2.2.0	18 February 2009	Official release. Event model updated.
	2.2.1	6 March 2009	Fixed reply confirmation bug. Added the ability to check the account that was used to perform messaging actions in the action's event handler, e.g., checking for replies.
	2.3.0	11 March 2010	Added synchronous messaging functionality. Updated phone number validation.

2. Audience

This document is intended for Microsoft .NET developers who wish to write applications that are able to send and receive SMS messages and voice messages.

3. Introduction

The **MessageMedia** API .NET Component has been developed to allow programmers to quickly and seamlessly integrate the **MessageMedia** Two-Way Messaging Platform (see Section 4) into their applications.

The API .NET Component is designed to be as developer-friendly as possible. It reduces the complexity of dealing with:

- Internet protocol
- **MessageMedia** propriety messaging protocol

By wrapping all the necessary communication protocols the API .NET Component allows you, the developer, to focus on solving bigger problems and developing more scalable solutions. By implementing an event-driven model the API .NET Component allows you to manage events in a scalable, structured way.

The **MessageMedia** API .NET Component is written in C# .NET allowing seamless integration with your .NET applications. The component is designed to be used in a Microsoft Windows programming environment that supports the .NET framework versions 2.0 and above.

Tested or known-to-work languages/environments are:

- ✓ ASP .NET
- ✓ Visual Basic .NET
- ✓ Visual C++ .NET
- ✓ Visual C# .NET

4. Messaging Platform

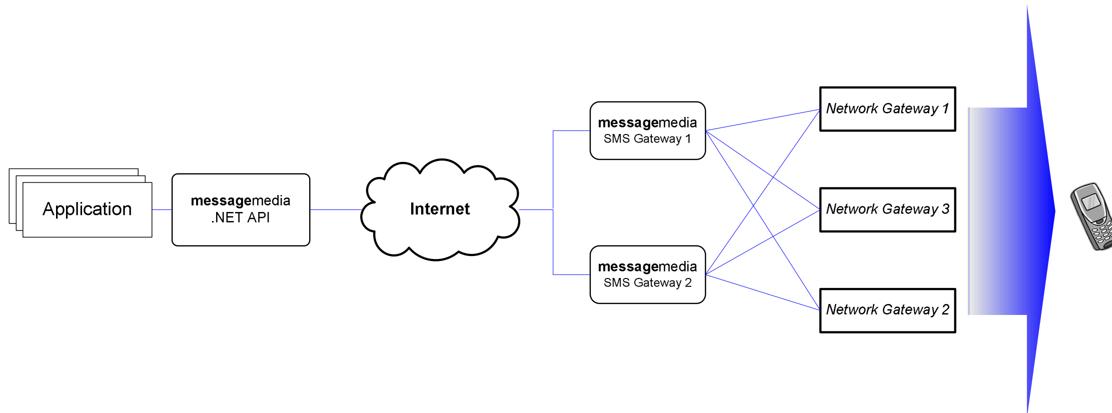
MessageMedia infrastructure delivers carrier grade reliability and feature rich service. **MessageMedia**'s communication infrastructure delivers the following services:

- **High Availability**
  There are multiple messaging gateways housed at separate sites. Each gateway is supported by multiple levels of redundancy (refer to Service Level Agreement guaranteeing 100% uptime).

- **Multiple-Carrier Access**
  MessageMedia provides intelligent carrier-switching, that routes messages through different carriers to maximise efficiency and service.

- **Service Features**
  Service features provided by the **MessageMedia** Messaging Platform include:
  - **Two-way Messaging**
    Send messages to any network and receive replies back to your application.
  - **Message Tagging**
    A unique feature of two-way messaging which tells your application exactly which outbound message corresponds to a given reply (important for transactional messaging applications).
  - **Dedicated Inbound Numbers**
    Provides your application with the ability to receive Mobile Originated SMS (also known as MO SMS or Incoming SMS). This enables your customers or workforce to send a message to a number that your application can respond to.
  - **Delivery Reporting**
    Enables tracking of messages, showing your application the exact time that each message you send is delivered on the handset. This provides your application with an audit trail and enables escalation in the event of non-delivery.
  - **Control of Validity Period**
    Allows you to specify exactly how long an outbound message will be valid on the mobile network. Once sent, the mobile network will continue to attempt delivery until the validity period expires.
  - **Message Splitting**
    Allows messages longer than 160 characters to be automatically split and sends the sub-messages in sequence at 30 second intervals.
  - **Premium SMS**
    Allows you to deliver SMS content that consumers are willing to pay a premium for. Premium SMS includes Mobile Originating (MO), where consumers are charged a tariff for each SMS message they send to your premium line, and Mobile Terminating (MT), where consumers are charged a tariff for each SMS message they receive from your premium line.
  - **API Web Push**
    Facilitates real-time delivery of inbound SMS direct to a URL, as opposed to ‘polling’, where messages are queued and then disseminated on request. This functionality is ideal for organizations that need replies from mobile staff or customers, delivered instantly. *Please note that this service is not compatible with the API .NET Component. Please contact **MessageMedia** for more details.*

The following diagram illustrates the communication lines between your application which uses the **MessageMedia** API .NET, the **MessageMedia** gateways, and the mobile network.



5. Requirements

To be able to use the **MessageMedia** API .NET Component you will need to have the following:

- Windows 2000 SP3 or later (including Windows XP and Windows 2003 Gateway)
- Internet connection (through HTTP port 80 or 443)
- Microsoft .NET Framework 2.0 or later

6. Constraints

The **MessageMedia** .NET is subject to the following constraints:

- **Reply Checking Interval**
  The reply checking interval is required to be greater than or equal to **one minute**. The property `Messaging.MessageController.Settings.AutoSendWaitTime`, which determines the automatic reply checking/message sending interval, cannot be set to less than one minute. Please note that due to network and computational limitations **MessageMedia** reserves the right to increase or decrease the minimum interval length.
  *A real-time inbound SMS solution is also available from **MessageMedia**. Please contact us for additional information.*

7. Using the API .NET Component

7.1 Installation

Please follow the instructions below to install **MessageMedia** API .NET Component:

- Download the MessageMedia API .NET Component installer package from http://www.message-media.com.au
- Launch the installer
- Follow the installer instructions to install the component, examples and documentation

- Depending on where you intend to use the component:

    - **Visual Studio .NET languages**
        - Launch your Microsoft Visual Studio .NET IDE
        - Open or create your project
        - Choose menu Project | Add Reference…
        - Click on the Browse button and select the **MessageMedia**.dll
        - Click OK to add the **MessageMedia** assembly into Solution Explorer
        - You are now ready to code in your preferred language

    - **ASP.NET web gateway environment**
        - Create a virtual directory pointing to:<destination>\aspVBTest folder where <destination> is where you install the component
        - Use your favourite web design software to create ASPX document in the above folder
        - Alternatively, you can copy the M4USMS.dll found in <destination>\aspVBTest\bin to your ASP.NET <web root>\bin directory

- Your installation is now complete and ready to be used. Please check the examples provided and explore this user manual. If you have any support questions after reading through the available documentation please contact us.

7.2 Setting up your Connection

Before any messages can be sent or received the connection to the **MessageMedia** gateway must first be configured. The gateway connection settings are all held by the `Messaging` singleton. The `Messaging` singleton can be accessed by calling `MessageMedia.Messaging.MessageController`. Here, `MessageMedia` is the namespace, `Messaging` is the singleton class and `MessageController` is the single instance of the `Messaging` class.

Setting up a connection with the **MessageMedia** gateway involves the following steps:

1. Configuring Gateway Settings
2. Configuring Proxy Settings (Optional)
3. Configuring Account Settings

7.2.1 Configuring Gateway Settings

The configuration of the gateway options determines two things:

1. How your application will communicate with the **MessageMedia** gateway
2. How the **MessageMedia** gateway will treat messages that are sent from your application

The gateway settings determine how your application will communicate with the **MessageMedia** gateway through options such as security (http or https?) and Internet timeout. The gateway settings also determine how the **MessageMedia** gateway will treat messages that are sent from your application. It does this through options such as auto-split (whether to automatically split long messages into multiple messages), batch size and delivery reporting.

The gateway settings are accessed via the property `MessageMedia.Messaging.MessageController.Settings`. The following code snippet gives an example of how to configure the gateway settings.

```
using MessageMedia;

…


// Use secure HTTP protocol.
Messaging.MessageController.Settings.SecureGateway = true;
// Set the Internet timeout for 60 seconds.
Messaging.MessageController.Settings.TimeOut = 60;
// Set the batch size (number of messages to be sent at once) to 200.
Messaging.MessageController.Settings.BatchSize = 200;
// Perform an automatic send/receive every minute.
Messaging.MessageController.Settings.AutoSendWaitTime = new TimeSpan(0, 1, 0);
// Enable delivery reporting to allow message tracking.
Messaging.MessageController.Settings.DeliveryReport = true;
```

7.2.2 Configuring Proxy Settings

If the application you are developing is to be run on a network that requires network traffic to go through a proxy gateway then the proxy settings will need to be configured. If no proxy settings are explicitly configured the API .NET Component will use Internet Explorer's proxy settings.

The proxy settings are held in the property `MessageMedia.Messaging.MessageController.Settings.Proxy`. The following code snippet gives an example of how to set up the proxy settings.

```
using MessageMedia;

…

// Use a proxy gateway.
Messaging.MessageController.Settings.Proxy.UseProxy = true;
// Set the host name of the proxy gateway to use.
Messaging.MessageController.Settings.Proxy.HostName = "proxy.mydomain.com";
// Set the port number that the proxy gateway listens to.
Messaging.MessageController.Settings.Proxy.Port = 80;
// Don't use default credentials.
Messaging.MessageController.Settings.Proxy.UseDefaultCredentials = false;
// Use the following username to authenticate with the proxy.
Messaging.MessageController.Settings.Proxy.UserName = "myusername";
// Use the following password to authenticate with the proxy.
Messaging.MessageController.Settings.Proxy.Password = "mypassword";
```

7.2.3 Configuring Account Settings

Before your application can send or receive messages you must first configure your account settings. Account settings are stored globally in the property `MessageMedia.Messaging.MessageController.UserAccount`. The account settings are sent to the gateway so that the gateway can authenticate you each time your application makes a request. The following code snippet gives an example of how to set up the account settings.

```
using MessageMedia;
using MessageMedia.Common;

…

// Set the username of the account holder.
Messaging.MessageController.UserAccount.User = "myusername";
// Set the password of the account holder.
Messaging.MessageController.UserAccount.Password = "mypassword";

// Set the first name of the account holder (optional).
Messaging.MessageController.UserAccount.ContactFirstName = "David";
// Set the last name of the account holder (optional).
Messaging.MessageController.UserAccount.ContactLastName = "Smith";
// Set the mobile phone number of the account holder (optional).
Messaging.MessageController.UserAccount.ContactPhone = "0423612367";
// Set the landline phone number of the account holder (optional).
Messaging.MessageController.UserAccount.ContactLandLine = "0338901234";
// Set the contact email of the account holder (optional).
Messaging.MessageController.UserAccount.ContactEmail = "david.smith@email.com";
// Set the country of origin of the account holder (optional).
Messaging.MessageController.UserAccount.Country = Countries.Australia;
```

Note that only the username and password are mandatory. The rest of the account information serves to provide **MessageMedia** with the ability to contact customers in relation to service issues and other important information. You should also always specify your country of origin. If you do not it will default to Australia. The country of origin is used by the function `Messaging.MessageController.PhoneNumberIsValid` to validate and clean phone numbers.

This function should always be called before sending messages; it is used to remove any extraneous characters from the phone number and add the international prefix (if required).

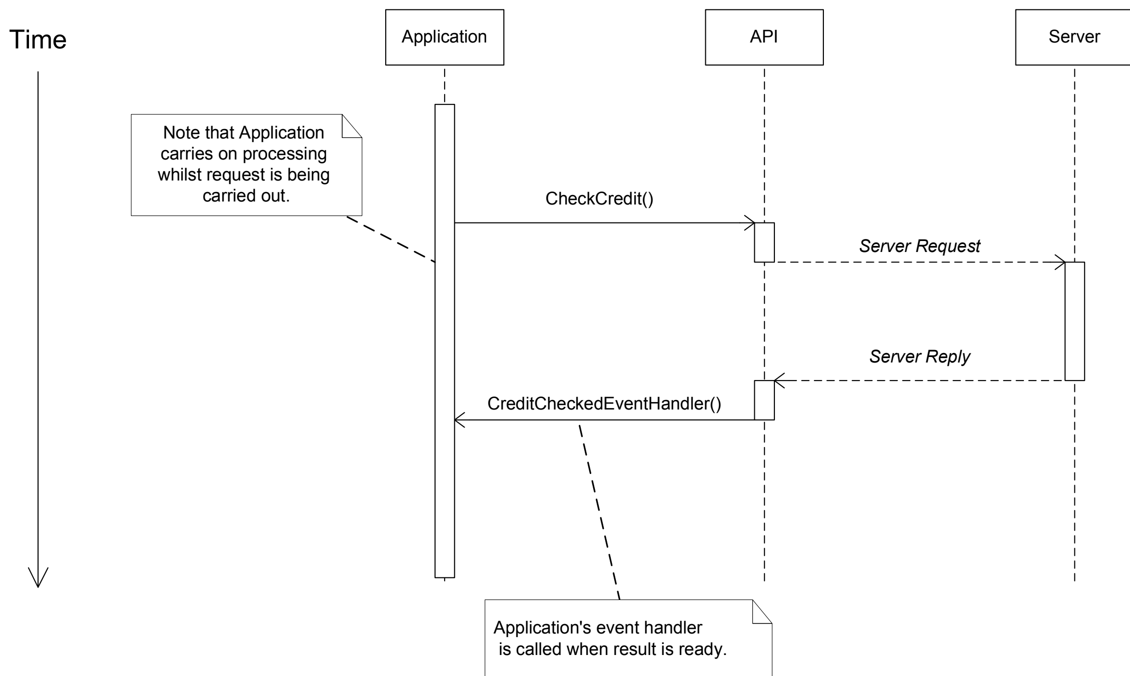
It is often useful to test whether the username and password are correct before you start trying to send or receive messages. The following code snippet shows how to test the username and password after they have been set as above.

```
using MessageMedia;

…

bool testOK = false;
try
{
    // Test the user account settings.
    Account testAccount = Messaging.MessageController.UserAccount;
    testOK = Messaging.MessageController.TestAccount(testAccount);
}
catch (Exception ex)
{
    // An exception was thrown. Display the details of the exception and return.
    string message = "There was an error testing the connection details:\n" +
                         ex.Message;
    MessageBox.Show(this, message, "Connection Failed", MessageBoxButtons.OK);
    return;
}


if (testOK)
{
    // The user account settings were valid. Display a success message
    // box with the number of credits.
    int balance = Messaging.MessageController.UserAccount.Balance;
    string message = string.Format("You have {0} message credits available.",
                                   balance);
    MessageBox.Show(this, message, "Connection Succeeded", MessageBoxButtons.OK);
}
else
{
    // The username or password were incorrect. Display a failed message box.
    MessageBox.Show(this, "The username or password you entered were incorrect.",
                    "Connection Failed", MessageBoxButtons.OK);
}
```

7.3 Event Handling

All messaging actions (e.g. sending a message, receiving a message, checking your credit) provided by the API .NET Component are available as both synchronous and asynchronous calls. For example to check replies asynchronously the method `Messaging.MessageController.CheckReplies()` would be called. To check replies synchronously `Messaging.MessageController.CheckRepliesSynchronous()` would be called. This documentation generally refers to the asynchronous versions of messaging calls, but remember that all messaging calls may also be made synchronously. Please see the API reference document that came with this API installation for full details.

Asynchronous messaging means that when, for example, you call `Messaging.MessageController.CheckReplies()` to check for and download reply messages the function does not wait until replies are downloaded before it returns. Instead it sends a request to the **MessageMedia** gateway asking for any reply messages and then returns straight away, allowing your application to carry on processing. When the gateway responds with any replies the API .NET Component will raise an event and call a method—known as an *event handler*—that you write. The following diagram uses the `CheckCredit()` API call to illustrate the sequence of events that leads to an event handler being called.



So, if your application will be using asynchronous messaging you will first need to set up event handlers. If your application will be calling the synchronous variants of the messaging functions you do not need to read this section.

The API .NET Component was designed according to an event-driven model. This makes your job as a developer much easier and also makes your code simpler to structure since you don't need to continually check for events that you are interested in. The following code snippet gives an example of how to attach event handlers.

```
using MessageMedia;
using MessageMedia.Events;
using MessageMedia.Diagnostics;

…

// Set up the event handlers.
Messaging.MessageController.SendMessagesComplete += new
```

```
    SendMessagesCompleteEventHandler(MessageController_SendMessagesComplete);
Messaging.MessageController.CheckRepliesComplete += new
    CheckRepliesCompleteEventHandler(MessageController_CheckRepliesComplete);
Messaging.MessageController.CheckCreditComplete += new
    CheckCreditCompleteEventHandler(MessageController_CheckCreditComplete);
Messaging.MessageController.ConfirmRepliesComplete += new
    ConfirmRepliesCompleteEventHandler(MessageController_ConfirmRepliesComplete);
Messaging.MessageController.MessagingStatusChanged += new
    MessagingStatusEventHandler(MessageController_MessagingStatusChanged);
Messaging.MessageController.UpdateUserDetailsComplete += new
   UpdateUserDetailsCompleteEventHandler(MessageController_UpdateUserDetailsComplete);
Messaging.MessageController.ErrorThrown += new
    ErrorThrownEventHandler(MessageController_ErrorThrown);
Logger.MessageLogged += new LoggerEventHandler(Logger_MessageLogged);
```

Here, `MessageController_SendMessagesComplete`, `MessageController_CheckRepliesComplete`, `MessageController_CheckCreditComplete`, `MessageController_ConfirmRepliesComplete`, `MessageController_MessagingStatusChanged`, `MessageController_UpdateUserDetailsComplete`, `MessageController_ErrorThrown`, and `Logger_MessageLogged` are event handlers. Event handlers are simply methods that conform to a specific delegate, which are then attached to an event. For example in the case of receiving messages, `MessageController_CheckRepliesComplete` is the event handler (the method that you write); `CheckRepliesCompleteEventHandler` is the delegate that the event handler's method declaration must conform to; and `CheckRepliesComplete` is the event itself. To check for replies you must first call `Messaging.MessageController.CheckReplies()`. When this action is complete the API will raise the `CheckRepliesComplete` event to notify your application. If any replies were downloaded they will be passed to your event handler in the event arguments. Reply handling is dealt with in more detail in Section 7.5.

Each of these events is documented in detail in the API Reference.

7.4 Sending a Message

Basic message sending consists of three steps:

1. Create a message object that contains the destination phone number and the message text
2. Add the message object to the global outbox queue
3. Call `Messaging.MessageController.SendMessages()` to send all messages that are waiting in the outbox

The following code snippet gives an example of how to send a single message.

```
using MessageMedia;
using MessageMedia.SMS;

…

// The destination phone number.
```

```
string phoneNumber = "0412345678";
// The text of the message to send.
string messageText = "Hey do u want 2 grab dinner 2nite?";

// Create a SMSMessage object.
SMSMessage smsObject = new SMSMessage(phoneNumber, messageText);

// Add the message to the outbox queue.
Messaging.MessageController.AddToQueue(smsObject);

// Send the message now.
Messaging.MessageController.SendReceiveNow();
```

Messages may also be sent in batches. Batches are simply groups of messages. Messages may be sent in a single batch, bypassing the outbox by calling `Messaging.MessageController.SendMessageBatch`. Batches of messages are stored in a queue object called `MessageQueue`. The outbox is simply a `MessageQueue` object that is owned by the `Messaging` singleton and may be accessed globally. The following code snippet shows how to create your own message queue and then send all the messages in the queue as a single batch.

```
using MessageMedia;
using MessageMedia.Common;
using MessageMedia.SMS;

…

// Create a message queue.
MessageQueue batch = new MessageQueue();
// Add some SMS messages to the queue.
batch.Add(new SMSMessage("0423612367", "hello 1"));
batch.Add(new SMSMessage("0423612367", "hello 2"));
batch.Add(new SMSMessage("0423612367", "hello 3"));

// Send the batch of messages.
Messaging.MessageController.SendMessageBatch(batch);
```

Messages that are queued in the outbox will remain unsent until the application calls `Messaging.MessageController.SendMessages()`. When `SendMessages` is called messages are sent from the outbox in batches; the number of messages included in each batch is determined by determined by the value of the property `Messaging.MessageController.Settings.BatchSize`. The default value of this property is 100 which is generally fine for most internet connections. If you have a slower internet connection and you start receiving internet timeout errors when sending messages try setting the batch size lower. If, on the other hand, you have a high speed internet connection you can set this value higher to send more messages to the gateway at once.

When all messages have been sent the event `SendMessagesComplete` is raised to notify your application that the send messages action has completed. When this event is raised your send messages event handler will be called (if you attached one as described in Section 7.3) and the

arguments passed will contain the messages that were sent in that batch. For example, let's say there are 400 messages in the outbox and the batch size is set to 100. The `SendMessagesComplete` event will be raised 4 times, each time reporting that a batch of 100 messages has been sent to the gateway. After the last time the `SendMessagesComplete` event is raised there will be no messages left in the outbox.

7.5 Receiving a Message

Every message that is sent by your application is assigned an identification number. You may assign the identification number yourself or you can let the API handle it for you. When someone replies to a message the reply message is assigned the same identification number as the message that the sent message it is in response to. This is how the system knows matches replies and delivery reports to specific sent messages.

To receive a reply message your application must first attach an event handler to the `CheckRepliesComplete` event. The event handler method must be of the form:

```
void MessageController_CheckRepliesComplete(object sender,
                                            CheckRepliesCompleteEventArgs e)
{
    …
}
```

The following code shows how to attach the event handler to the `CheckRepliesComplete` event.

```
Messaging.MessageController.CheckRepliesComplete +=
    new CheckRepliesCompleteEventHandler(MessageController_CheckRepliesComplete);
```

Once the event handler has been attached the API .NET Component will call your method when new reply messages have been downloaded. Replies can be checked for and downloaded manually by calling `Messaging.MessageController.CheckReplies()`. This function contacts the MessageMedia gateway and downloads any reply messages that are waiting. Once the action is complete your event handler will be called. The `MessageMedia.Events.CheckRepliesCompleteEventArgs` parameter stores the downloaded reply messages in a list. In the above example the reply message list can be accessed as `e.Messages`. If delivery reporting has been enabled delivery reports will also be downloaded as part of the check reply process. In the above example the list of delivery reports can be accessed as `e.DeliveryReports`. Delivery reporting is covered in more detail in Section 7.6.

To save having to call `CheckReplies` each time you wish to check for and download replies and delivery reports you can enable the auto- receive feature. When this feature is enabled the system will automatically check for and download replies at regular intervals. To enable this feature set the

property `Messaging.MessageController.AutoReceive` to true. The default interval for this feature is 5 minutes. The default interval can be changed by modifying the property `Messaging.MessageController.Settings.AutoReceiveWaitTime` to a different value. This value cannot be set to less than one minute.

To prevent excessive loads on the gateway replies cannot be checked for more than once per minute per account. If `CheckReplies` is called more than once per minute with the same account it will return false; it will not execute the action and the `CheckRepliesComplete` event will not be generated. You may obtain the amount of time required to wait before replies can be checked for by calling `Messaging.MessageController.GetCheckReplyTimeToWait()`.

7.6 Obtaining Delivery Reports

Delivery reports may be used to determine if and when a message arrives on a recipient's handset. Receiving delivery reports may incur an additional charge, for example in Australia each delivery report is charged at 10 cents per report. Delivery reporting is quite carrier-specific and often different results will be obtained depending on the carrier that the recipient is signed with. Most carriers will send a delivery report indicating that the sent message was delivered to the recipient's handset at a specific time.

To enable delivery reporting set `Messaging.MessageController.Settings.DeliveryReport` to true. Delivery reports are downloaded from the gateway as part of the check reply process. When the event handler for the `CheckRepliesComplete` event is called the list of downloaded delivery reports are available in the `MessageMedia.Events.CheckRepliesCompleteEventArgs` parameter. Like reply messages, the delivery report for a specific sent message will have the same message ID as the sent message. The `DeliveryReport` object stores the delivery status of the sent message in the `Status` property and the delivery time in the `Received` property. The use of delivery reporting is shown in detail in the sample application provided with the SDK.

7.7 Checking your Account Credit

Like sending messages, receiving messages, and most actions in the API .NET Component, credit checking is asynchronous and involves event handling.

To check your credit balance you must first attach an event handler to the `CheckCreditComplete` event. The event handler method must be of the form:

```
void MessageController_CheckCreditComplete(object sender,
                                           CheckCreditCompleteEventArgs e)
{
    …
}
```

The following code shows how to attach the event handler to the `CheckCreditComplete` event.

```
Messaging.MessageController.CheckCreditComplete +=new
```

```
    CheckCreditCompleteEventHandler(MessageController_CheckCreditComplete);
```

Once the event handler has been attached the API .NET Component will call your method when the credit check action is complete. To check your credit call the method `Messaging.MessageController.CheckCredit()`. Since this operation is asynchronous the function will send a credit-check request to the **MessageMedia** gateway and return straight away. When the gateway responds with your available credit balance your credit-check event handler will be called so that you can handle the event.

Your credit balance is checked each time you send messages so that your balance is updated without needing to explicitly call `CheckCredit`. After each batch of messages are sent to the gateway the `CheckCreditComplete` event will be raised to notify your application of the updated balance of your account. The most recently obtained balance can always be obtained from `Messaging.MessageController.UserAccount.Balance`.

8. Contact Us

Feedback is always welcome! If anything in this document is unclear please do not hesitate to contact us.

Australia

Telephone: 1800 155 228
Email: support@message-media.com.au

New Zealand

Telephone: 0800 68 69 64
Email: support@message-media.co.nz

USA

Telephone: 1866 884 8611
Email: support@message-media.com

UK

Telephone: 0808 234 4874
Email: support@message-media.com