devCentral

Developing Applications for Pocket PC and GPRS/EDGE

devCentral White Paper

Document Number	**12588**
Revision	**2.0**
Revision Date	**10/15/03**

AT&T Wireless Developer Program

AT&T Wireless

Legal Disclaimer

This document and the information contained herein (collectively, the "**Information**") is provided to you (both the individual receiving this document and any legal entity on behalf of which such individual is acting) ("**You**" and "**Your**") by AT&T Wireless Services, Inc. ("**AWS**") for informational purposes only. AWS is providing the Information to You because AWS believes the Information may be useful to You. The Information is provided to You solely on the basis that You will be responsible for making Your own assessments of the Information and are advised to verify all representations, statements and information before using or relying upon any of the Information. Although AWS has exercised reasonable care in providing the Information to You, AWS does not warrant the accuracy of the Information and is not responsible for any damages arising from Your use of or reliance upon the Information. You further understand and agree that AWS in no way represents, and You in no way rely on a belief, that AWS is providing the Information in accordance with any standard or service (routine, customary or otherwise) related to the consulting, services, hardware or software industries.

AWS DOES NOT WARRANT THAT THE INFORMATION IS ERROR-FREE. AWS IS PROVIDING THE INFORMATION TO YOU "AS IS" AND "WITH ALL FAULTS." AWS DOES NOT WARRANT, BY VIRTUE OF THIS DOCUMENT, OR BY ANY COURSE OF PERFORMANCE, COURSE OF DEALING, USAGE OF TRADE OR ANY COLLATERAL DOCUMENT HEREUNDER OR OTHERWISE, AND HEREBY EXPRESSLY DISCLAIMS, ANY REPRESENTATION OR WARRANTY OF ANY KIND WITH RESPECT TO THE INFORMATION, INCLUDING, WITHOUT LIMITATION, ANY REPRESENTATION OR WARRANTY OF DESIGN, PERFORMANCE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, OR ANY REPRESENTATION OR WARRANTY THAT THE INFORMATION IS APPLICABLE TO OR INTEROPERABLE WITH ANY SYSTEM, DATA, HARDWARE OR SOFTWARE OF ANY KIND. AWS DISCLAIMS AND IN NO EVENT SHALL BE LIABLE FOR ANY LOSSES OR DAMAGES OF ANY KIND, WHETHER DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, SPECIAL OR EXEMPLARY, INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, LOSS OF GOODWILL, COVER, TORTIOUS CONDUCT OR OTHER PECUNIARY LOSS, ARISING OUT OF OR IN ANY WAY RELATED TO THE PROVISION, NON-PROVISION, USE OR NON-USE OF THE INFORMATION, EVEN IF AWS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES OR DAMAGES.

Revision History

Date	Revision	Description
03/14/03	1.1	First release of this document for Pocket PC 2002.
09/05/03	1.2	New devCentral template applied to document.
09/15/03	1.3	Links updated throughout document.
10/15/03	2.0	Document updated for Pocket PC 2003.

Table of Contents

Table of Contents

Figures

Tables

1. Introduction

This paper explains how developers can design applications that operate on the Microsoft Pocket PC platform and that communicate using the AT&T Wireless General Packet Radio Service (GPRS)/Enhanced Data Rates for GSM Evolution (EDGE) network.

The Pocket PC platform, based on the Microsoft Windows CE operating system, is an ideal complement for wireless communications as it offers significant computing power in a small convenient form factor, enabling a variety of communications-oriented applications that can enhance productivity.

Since Microsoft and other parties provide thorough documentation on developing Pocket PC applications, this paper emphasizes those aspects that are unique to wireless networking.

This version of the paper emphasizes development for the Pocket PC 2003 platform. The first version of this paper released in March 2003 covered the Pocket PC 2002 platform. Pocket PC 2003 introduces a number of new features of interest to developers including:

- The use of the Windows CE 4.2 operating system
- The Inclusion of the .NET Compact Framework in ROM
- The ability with integrated devices (based on Pocket PC Phone Edition) to maintain GPRS/EDGE connections even while suspended
- Enhancements to Pocket Internet Explorer
- Enhanced SMS capability that simplifies application access to SMS messages.
- Bluetooth support with a new API
- Enhanced security features, including support for L2TP and IPSec.

For a thorough discussion of the new features in Pocket PC 2003, refer to Microsoft's Whitepaper, *What's New for Developers in Windows Mobile 2003-based Pocket PC,* referenced in Section 1.3 of this document.

Pocket PC 2003 also benefits end-users with zero-configuration Wi-Fi support, an improved version of Outlook, a new version of Media Player, and improved keyboard support. However, these features should not affect wireless-application developers.

This paper begins with an overview of developing for the Pocket PC, and then discusses types of devices, connection management, power management, security, voice call control, and short message service and SIM access. It then explains software development tools in detail, followed by programming considerations. It finally presents a sample application in detail using the two primary development environments.

1.1 Audience

This paper has been developed for independent software vendors, corporate developers, and system integrators engaged in application development. The paper assumes a working knowledge of GPRS and EDGE, and reasonable knowledge of Windows programming, as well as some knowledge of Pocket PC.

1.2 Contact Information

E-mail any comments or questions regarding this document to developer.program@attws.com. Please reference the title of this document in your e-mail.

Authors: Mark Hotopp, Farzeen Mohazzabfar, support@QualityInMotion.com, and Peter Rysavy, http://www.rysavy.com.

1.3 Resources

1.3.1 AWS Resources

AT&T Wireless Communication Manager Developer Reference
http://www.attwireless.com/developer/technologies/awsCommunicationManager.jhtml

1.3.2 Microsoft Resources

What's New for Developers in Windows Mobile 2003-based Pocket PC,
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnppc2k3/html/winmob03.asp

eMbedded Visual C++ 4.0 and Windows CE .NET, Microsoft White Paper,
http://msdn.microsoft.com/vstudio/device/embedded/evcandcenet.asp

Windows CE 3.0: Application Programming, Nick Grattan, Marshall Brain, October 2000, Prentice Hall, ISBN – 0130255920.

Pocket PC Network Programming, Steve Makofsky, July 2003, Addison-Wesley, ISBN – 0-321-13352-8

Microsoft Pocket PC Technical Articles,
http://www.microsoft.com/windowsmobile/resources/technicalarticles/pocketpc/default.mspx

Microsoft: *What's New for Developers in Windows Mobile 2003-based Pocket PC*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnppc2k3/html/winmob03.asp

Microsoft article: *Comparing Web Controls and Mobile Controls.*
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mwsdk/html/mwconcomparingwebcontrolsmobilecontrols.asp

Microsoft article *Build Better Applications with SQL Server CE 2.0*.
http://www.microsoft.com/sql/CE/default.asp

Microsoft MSDN Library, October 2003.
http://msdn.microsoft.com/library/default.asp

Windows Mobile Developer Portal:
http://www.microsoft.com/windowsmobile/developer

Microsoft Developer Subscriber Downloads:
http://msdn.microsoft.com/subscriptions/downloads/default.asp

Microsoft Windows Platform SDK for Pocket PC 2002, Help files, Software Developer Kit, January 2002.

Microsoft Windows Platform SDK for Pocket PC 2003, Help files, Software Developer Kit, April 2003.

Microsoft eMbedded Visual Tools 3.0, Help files, Software Developer Kit

Microsoft White Paper: *Smart Device Programmability*, http://msdn.microsoft.com/vstudio/device/smartdev.asp.

1.3.3 3GPP Resources

3GPP TS 01.61, Technical Specification, *General Packet Radio Service (GPRS); GPRS Ciphering Algorithm Requirements*, http://www.3gpp.org/ftp/Specs/2003-06/R1999/01_series/0161-800.zip

3GPP TS 03.40, Technical Specification: *Technical Realization of the Short Message Service (SMS)*, http://www.3gpp.org/ftp/Specs/2003-06/R1998/03_series/0340-750.zip

3GPP TS 03.60, Technical Specification: *General Packet Radio Service (GPRS), Service Description*. http://www.3gpp.org/ftp/Specs/2003-06/R1998/03_series/0360-790.zip

3GPP TS 07.05, Technical Specification: *Use of Data Terminal Equipment—Data Circuit Terminating Equipment (DTE–DCE) Interface for Short Message Service (SMS) and Cell Broadcast Service (CBS)*, http://www.3gpp.org/ftp/Specs/2003-06/R1998/07_series/0705-701.zip

3GPP TS 07.07: Technical Specification, Third Generation Partnership Project: *AT Command Set for GSM Mobile Equipment (ME)*, http://www.3gpp.org/ftp/Specs/2003-06/R1998/07_series/0707-780.zip

3GPP TS 11.11, Technical Specification: *Specification of the Subscriber Identity Module—Mobile Equipment (SIM-ME) Interface*, http://www.3gpp.org/ftp/Specs/2003-06/R1999/11_series/1111-891.zip

1.3.4 Other Resources

ASP.NET Mobile Controls. http://www.asp.net/mobile/intro.aspx

MSDN Mobile and Embedded Developer Center: http://www.msdn.com/mobility

Quality in Motion: Software research, tool analysis, and development of the demonstration applications: e-mail: farzeen@qualityinmotion.com

1.4 Terms and Acronyms

Table 1 defines terms and acronyms used in this document.

Table 1: Terms and Acronyms

Term or Acronym	Definition
API	Application Program Interface
APN	Access Point Name
C#	(Pronounced C sharp) An object-oriented computer programming language from Microsoft that enables programmers to quickly build a wide range of applications for the Microsoft .NET platform.
CE	Compact Edition
cHTML	Compact HTML
CSV	Comma Separated Value
DNS	Domain Name System
EDGE	Enhanced Data Rates for GSM Evolution
FTP	File Transfer Protocol
GEA	GPRS Encryption Algorithm
GGSN	GPRS Gateway Support Node
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
GUID	Global Unique Identifier
HLR	Home Location Register
HTML	Hypertext Markup Language
IDE	Integrated Design Environment
IL	Intermediate Language
IP	Internet Protocol

Term or Acronym	Definition
IPSec	Secure IP
IR	Infrared
L2TP	Layer 2 Tunneling Protocol
MFC	Microsoft Foundation Class
MS	Mobile Station (mobile computer plus communications device)
MSIL	Microsoft Intermediate Language
MSISDN	Mobile Subscriber Integrated Services Digital Network
NAT	Network Address Translation
NDIS	Network Driver Interface Specification
OID	Object Identifier
PCMCIA	Personal Computer Memory Card International Organization
PDA	Personal Digital Assistant
PDP	Packet Data Protocol
PPTP	Point to Point Tunneling Protocol
SD	Secure Digital
SDK	Software Developer Kit
SGSN	Serving General Packet Radio Service (GPRS) Support Node
SIM	Subscriber Identity Module
SMS	Short Message Service
SQL	Structured Query Language
SSL	Secure Sockets Layer
TAPI	Telephony API
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USSD	Unstructured Supplementary Service Data
VPN	Virtual Private Network
WAP	Wireless Application Protocol
WML	Wireless Markup Language
XML	Extensible Markup Language

2. Overview: Developing for Pocket PC

Developing applications for the Pocket PC is similar in many respects to developing applications for Microsoft Windows for desktop and notebook platforms. Development tools are similar, as are most of the APIs, and the platform itself provides numerous capabilities and comes standard with a variety of applications, including:

- Outlook for contacts, e-mail, notes, and calendar
- Synchronization with host (e.g., desktop) computers via Microsoft ActiveSync
- Pocket Word, Pocket Excel, and Windows Media Player
- Other capabilities, including:
  - A multi-threaded and multi-tasking application environment
  - A browser that supports HTML, XML/XSL, WML, cHTML, Jscript, and SSL
  - Graphics support with up to 65,536 colors
  - Interfaces for Short Message Service (SMS), Subscriber Information Module (SIM) information, and telephony control via the Telephony API (TAPI)
  - Expansion capabilities with options for Compact Flash, Secure Digital (SD), PCMCIA, and multimedia card

There are some significant differences in working with the Pocket PC. Specifically, applications must take into account memory constraints, as well as user interface constraints. Today's Pocket PC comes standard with 32 MB or 64 MB of storage for programs and data, though memory can be added easily using the SD or Compact Flash slots.

Developers must consider the user interface carefully. Current Pocket PC devices typically have a resolution of 240 X 320 pixels (76,800 total), compared to a typical resolution of 1024 X 768 pixels (786,432 total) for notebook platforms. This is only about ten percent of the viewing area of larger platforms. Fortunately, the Pocket PC display, combined with good graphical capabilities, is sufficient for a wide range of applications. The screen can comfortably display sixteen lines of text with six or seven words per line.

Another consideration is the ability to enter information. Without a keyboard, the amount of data a user can enter (and the speed at which it can be entered) is limited, and must be accounted for by the developer to avoid a frustrating user experience. For example, the stylus-based user interface is not well suited for typing in long messages. However, it works very well for making selections from predefined lists or entering small amounts of data, perhaps two to ten words. Keyboard options are available for most Pocket PCs, but diminish its portability.

The smaller display of the Pocket PC actually makes the platform well suited for GPRS/EDGE communications, as it only takes a small amount of data to populate the screen, which can happen quickly over a GPRS connection. Furthermore, the ability of GPRS/EDGE to deliver IP-based networking over extended geographic areas is highly complementary with the portable nature of the device.

There are three fundamental approaches for developing applications for Pocket PC 2003:

1. Using Microsoft eMbedded Visual C++ 4.0 (Service Pack 2).

   Note: eMbedded Visual Basic is no longer supported in Pocket PC 2003. It has been superseded by Visual Basic .NET. However, existing eMbedded Visual Basic applications continue to run for backward compatibility as long as the Pocket PC 2003 device contains the Visual Basic runtime DLLs. For more information on how to move eMbedded Visual Basic applications to Visual Basic .NET visit: http://msdn.microsoft.com/library/en-us/dnppc2k3/html/fromemb.asp

2. Using Visual Studio .NET 2003 and The Microsoft .NET Compact Framework.
3. Using ASP.NET Mobile Controls (formerly the Microsoft Mobile Internet Toolkit)

To illustrate the steps involved in developing wireless applications for Pocket PC, this paper describes a simple application that targets the Pocket PC platform and uses GPRS/EDGE. This is done for both eMbedded Visual C++ 4.0 and .NET Compact Framework environments. This paper also discusses the pros and cons of each development environment.

The sample application is a stock-quote application that allows users to enter a stock ticker symbol. Using a GPRS/EDGE connection, the application makes a request to a stock quote service to retrieve the current price. The application uses the Yahoo Comma-Separated Value (CSV) quote service, but other quote providers could easily be substituted.

The first approach, eMbedded Visual C++ 4.0, is very similar to building a desktop application using Visual C++. The sample application is a Microsoft Foundation Class (MFC) dialog based application. This paper discusses how to create the project, add code to perform user interface and networking operations, how to build the application for various targets, and how to debug the application. If you have developed applications using Visual C++ already, you will notice how similar building this application is. For size and speed considerations, the sample application could also be written using the Windows CE API (which is similar to Win32 programming).

The second approach uses Visual Studio .NET 2003 to illustrate how an application can be created for the .NET Compact Framework. The .NET Compact Framework application differs significantly in several respects. First, it is compiled to Microsoft Intermediate Language (MSIL), the intermediate language created in the .NET platform, which is a high level machine code that is device independent. The MSIL code is translated at runtime to the target device machine code. There are some benefits and drawbacks to this method, which are discussed later.

The third approach, ASP.NET Mobile Controls, is based on a browser approach. ASP.NET allows you to create content that can be rendered on a variety of devices, including the Pocket PC and phones with micro browsers (e.g., WAP). This paper does not show a sample application for this third approach but does discuss the approach and the pros and cons of this approach in the section, ASP.NET. To find out more about ASP.NET Mobile Controls, you can also visit: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mwsdk/html/mwconcomparingwebcontrolsmobilecontrols.asp.

A final approach, and one not covered in this paper, is Microsoft SQL Server 2000 Windows CE Edition (SQL Server CE) version 2.0. This approach allows applications that extend enterprise data management capabilities to Pocket PC devices and store and manipulate large amounts of data. Essentially, the Pocket PC operates as an extension of an enterprise database. This approach is well suited for scenarios where the device does not enjoy constant connectivity (as provided by GPRS/EDGE) and is only periodically synchronized.

To find out more about SQL Server CE, please visit: http://www.microsoft.com/sql/CE/default.asp .

An alternative to developing an application for the Pocket PC environment is to make it available via Microsoft Windows Terminal Services. In this approach, the application operates on a Windows Server at a central location. The Pocket PC uses the Terminal Services Client, which comes with the Pocket PC platform. With Terminal Services, the Terminal Server sends screen updates to the Terminal Services Client, and the client sends user input to the server. This approach is feasible using GPRS/EDGE connections.

The advantage of this approach is that applications can be maintained in a central location, and no software needs to be distributed for the mobile devices. One item to consider is that many existing applications assume a screen size larger than offered by the Pocket PC. To avoid inconvenient screen scrolling by the user, it may be best to reformat the display output to the parameters of the Pocket PC screen. To mitigate this need, the Pocket PC client offers an option to limit the size of the server desktop to fit on the screen.

Table 2 summarizes the tools that are available for the different platforms.

General Notes Applicable to Table 2:

Pocket PC = Operating System + Form Factor

Windows CE is the Operating System.

If you are going to build a .NET CF (Compact Framework) application, you should distribute the CF if you are targeting for Pocket PC 2000/2002

Table 2: Tools Versus Platforms

	Platforms		
Tools	**Pocket PC 2000 (CE Version**	**Pocket PC 2002/ Phone Edition (CE Version 3.1)**	**Pocket PC 2003/Phone Edition (CE.NET Version.4 2)**
Embedded Visual Tools 3.0 (See Note 1)	Yes	Yes	Yes (but you cannot deploy or debug)
Embedded Visual C++ 4.0 SP 2 (See Note 2)	No	No	Yes
Visual Studio .NET 2003	Yes (See Notes 3 and 4)	Yes (See Note 4)	Yes (See Note 4)
Includes Microsoft Connection Manager	No	Yes	Yes

Table Notes:

1. Includes Embedded Visual Basic and Embedded Visual C++ 3.0
2. Can only create native executables (Cannot create .NET applications)
3. According to: http://msdn.microsoft.com/chats/embedded/embedded_103101.asp
4. Can only create .NET CF (Compact Framework) applications (Native applications can be created via Embedded Visual C++ 4.0 SP2)

3. Types of Devices

This section briefly discusses the various device types, configurations, and their characteristics. The Windows Mobile platform currently offers PDAs, PDAs with phone capability, and Smartphones. Smartphones differ from PDAs in that they have a traditional mobile telephone keypad, do not rely on a stylus, and have a smaller display. This paper does not discuss Windows Mobile-based Smartphones and concentrates on the Pocket PC platform.

Note: Microsoft provides a separate SDK for the Smartphone.

PDAs that have an integrated phone capability are based on a version of Pocket PC called Pocket PC Phone Edition. Development tools are the same for both Pocket PC and Pocket PC Phone Edition. The difference is that the telephony and GPRS/EDGE features in Pocket PC Phone Edition are more tightly integrated. For example, users can tap on contacts in Outlook to initiate a phone call.

PDAs without phone capability can communicate over GPRS/EDGE using either a sleeve that accepts PC Card modems (with Compact Flash devices a future option), or via a tethered connection to a GPRS/EDGE capable mobile telephone. Tethering options include cables, Infrared (IR) or Bluetooth.

Note: Specific connectivity options (e.g. sleeves, Bluetooth, cables) will vary depending on the vendors of the PDA, modem, or mobile telephone.

Table 3 summarizes the available devices and their characteristics.

Table 3: Types of Windows CE Devices

Device	Connectivity	Comments
Pocket PC	Sleeve for PC Card, Compact Flash, tethered using cable, IR or Bluetooth	Any Pocket PC PDA (version 2002 or later) can be used with GPRS, though specific connectivity options may vary. Two-handed operation using a stylus.
Pocket PC 2002/2003 Phone Edition	Integrated GSM/GPRS/EDGE	AT&T Wireless offers the Siemens SX-56, which currently supports GSM/GPRS (not EDGE). Tight integration between platform and communications functions.
Smartphone	Integrated GSM/GPRS/EDGE	AT&T Wireless offers the Motorola MPx200, which is a GPRS device. Single-hand operation using phone keypad. No stylus. Details on developing for this platform can be found at the AT&T Wireless devCentral Web site at http://www.attwireless.com/developer/technologies/smartphone/.

Note: Companies such as Symbol Technologies have also developed integrated devices that combine a Pocket PC and GSM/GPRS/EDGE communications modules. While highly integrated, these devices are not necessarily based on Pocket PC 2003 Phone Edition.

Please refer to the following link on the AT&T Wireless Developer Web site for further information regarding specific Pocket PC devices that AT&T Wireless supports:
http://www.attwireless.com/developer/devices/PDAs/

4. Connection Management

A crucial aspect of GPRS/EDGE application development for the Pocket PC is managing the GPRS/EDGE connection. Section 3 discussed the different types of GPRS/EDGE devices. To the Pocket PC, the GPRS/EDGE device appears as either a modem or network device, depending on how the GPRS/EDGE device vendor has implemented their device. The Pocket PC provides networking (e.g., Network Driver Interface Specification—NDIS) and modem interfaces that allow the TCP/IP protocol stack to communicate with the GPRS/EDGE device.

For a GPRS/EDGE device to be able to communicate with the network, it must initiate what is called a Packet Data Protocol (PDP) Context. This results in the network assigning it an IP address and knowing the location of the device. Users can manually invoke a GPRS/EDGE connection, but in many instances, this will involve an extra step for the user. It is best if the application initiates a connection if a connection is not already established.

New to Pocket PC 2003 is the ability to maintain a GPRS/EDGE connection when the device is in standby mode. GPRS/EDGE data connections are instantly resumed after a completed voice call or once the device is turned back on. This greatly enhances the user experience with no additional development burden for the developer.

For devices with modem-based interfaces, an application can initiate a PDP context by issuing a modem AT command to the device as per the 3GPP Specification 07.07. However, it is programmatically simpler to work with higher level APIs that manage the connection.

Pocket PC 2003 provides a Connection Manager API that handles this function. This API is discussed in detail in the Section 12.1.4, Connection Manager. This API is available for integrated devices and system configurations where the GPRS/EDGE device can be controlled through the Pocket PC Settings/Connections/Connections mechanism.

With devices like the HP iPaq, that requires an add-on card in order to create a GPRS/EDGE connection, AT&T Wireless provides its own connection management utility called Communication Manager. The AT&T Wireless utility is a standalone application that provides system configuration, diagnostic information, troubleshooting support, connection management, and Web-traffic optimization. It also provides an API that can be used by other applications to create or test for a GPRS/EDGE connection.

Communication Manager is very similar to the Connection Manager built into Pocket PC 2002 and Pocket PC 2003. However, unlike the connection manager provided by Pocket PC, which has a C-call level API, the AT&T Wireless utility exposes a Windows messaging API. This means that an application must send Windows messages to the main window of the connection manager. The following is a list of all the functions that are available:

- **Connect:** Send this message in order to tell the connection manager to connect for you.
- Disconnect: Send this message to tell the connection manager to disconnect for you.
- **Get status:** Send this message in order to determine the current status of the connection.
- **Display:** Send this message to tell the connection manager to display itself.
- **Shutdown:** Send this message to tell the connection manager application to shut itself down.

Note: Currently, Communication Manager only supports Pocket PC 2002. It does not support Pocket PC 2002 Phone Edition, Pocket PC 2003, nor Pocket PC 2003 Phone Edition.

The principal advantage of using the AT&T Wireless API is that it supports some system configurations not supported by the Microsoft API. It also provides cross-platform support including non-Pocket PC devices. For more detailed information on how to program with the AT&T Wireless Communication Manager, refer to AT&T Wireless Communication Manager Developer Reference at the following link:

http://www.attwireless.com/developer/technologies/awsCommunicationManager.jhtml

Developers should be careful that their application does not make any assumptions about connection status, as a number of factors can control the connection. These include:

- **Network Factors:** The GPRS/EDGE network will terminate a PDP Context after four hours of inactivity or one hour out of coverage.
- **Signal Quality:** If the signal is very weak or absent, the modem itself may terminate the connection.
- **Power Management Settings:** The device will terminate the connection after the number of minutes specified in Settings/System/Power. Note for Pocket PC 2003 Phone Edition, the GPRS connection is maintained even in standby.
- **User Actions:** A user manually turning off the device will cause the connection to drop.
- **Other Applications:** Other applications may also be managing the connection. For example, Pocket IE will terminate the GPRS/EDGE connection after ten minutes of inactivity.

Since the application is not necessarily aware of the events listed above, it should always check the connection status before initiating communications. The application itself may also terminate a connection. An application can accomplish this by using the Connection Manager API.

Typically, this would only occur when the application is closed. However, since it is up to the application developer, any algorithm could be used. For example, the application can employ an inactivity timer. Given that billing is generally based on the volume of data communicated, and given that establishing the GPRS/EDGE connection (PDP context activation) takes five to ten seconds, the user will generally have the best experience when the connection is maintained.

As mentioned in the bulleted list above, power management settings on the device can cause connections to terminate. If the device is set to shut off after a specified period of inactivity, the connection will terminate. Unfortunately, Pocket PC does not consider an open socket the same as network activity. If your application requires a connection for long periods of time, regardless of user activity, you must address this issue. One simple option is for your documentation, or application, to prompt the user to change their power management settings. Another option is to change the settings when your application opens, and then revert to their original settings when your application closes. Alternatively, you could create a thread that performs some arbitrary background processing at preset intervals in order to keep the power management routines from activating. Doing so will obviously have power implications.

5. Power Management

Power management is a much more sensitive issue for Pocket PC development than for desktop development. Since the Pocket PC platform can shut off at any time for many reasons, a developer must consider this when designing an application.

Power being switched off has a much more profound effect on network-centric applications, because at one moment your code logic may assume a good connection is available, but the next moment the device is turned off. When the device is resumed, you may attempt to use that connection, which in most cases is now broken. This means your application must be written in a manner that allows it to robustly reestablish a connection or regain a resource at any point in time. If not, the user will be forced to terminate your application and then restart it in order for it to work properly again.

Note: Pocket PC 2003 Phone Edition can maintain GPRS/EDGE connections even when the unit is powered off. This can simplify network application development considerably.

Another consideration for applications that may be particularly power hungry, is the current power level of the device. If your application relies on high wattage peripherals or requires a lot of CPU and memory use, you may consider monitoring the device power. This can be done using the **CeGetSystemPowerStatusEx** function. When you determine that the power status is getting dangerously low, you can take appropriate action.

6. Security

When considering security, developers should take into account the security features provided by GPRS/EDGE technology, connectivity between the AT&T Wireless GPRS/EDGE network and the customer network, and the security features and mechanisms available within Pocket PC.

Air-interface ciphering (encryption) in GPRS/EDGE is similar to that of the GSM voice network. The encryption algorithm is called the GPRS Encryption Algorithm (GEA.) The strength of GEA is roughly equivalent to that of A5 used in GSM. The GPRS/EDGE network encrypts data communication between the modem and the portion of the network infrastructure called the Serving GPRS Support Node (SGSN). Beyond the SGSN, the data is not encrypted—however it does travel across a private network until it reaches the Gateway GPRS Support Node (GGSN), which is the gateway to external networks.

Beyond the GGSN, AT&T Wireless offers a number of options to secure the connection to customer networks. These options include the use of frame relay permanent virtual circuits and VPN connections for Internet-based connections. Refer to the AT&T Wireless devCentral Web site: http://www.attwireless.com/developer/ for more information.

The security model of the Pocket PC itself is very similar to that of the desktop versions. Microsoft makes many APIs available to aid the developer in creating secure applications. These are the same APIs available to the desktop developer, but with reduced functionality. All the major cryptographic functions are part of the Windows CE.NET 4.2 operating system.

The cryptographic API is the lowest-level API available for performing secure communications. It contains functions to perform the following operations:

- Hash data using standard algorithms such as SHA, MD5, and many others
- Get random data used for seeding or public key infrastructure
- Import and export keys of various types
- Encrypt and decrypt data using standard protocols such as DES3, RC2, RC4, and many others

- Perform standard authentication operations using protocols such as Kerberos, NTLM, and many others
- Extend hashing, encrypting, and authenticating protocols with custom providers that you may require

The HTTP Control API is a higher-level API available for use, but it is not specifically security related. This API will allow you to connect to HTTPS sites and securely transfer data to and from those sites. If the communications you are performing can be done with a Web server via SSL, then this is the simplest and fastest way to go.

There is also VPN support provided by the Pocket PC for the Point-to-Point Tunneling Protocol (PPTP) and the Layer 2 Tunneling Protocol (L2TP) in conjunction with IPSec. Previously on Pocket PC devices, only PPTP was available as part of the operating system. This can secure application communications without requiring any support from the application itself. In addition, many of today's VPN hardware and software vendors have Pocket PC-compatible clients available.

For more information on how to access VPN connectivity on the Pocket PC, see ***CM_VPN_Entries*** in the Pocket PC 2003 SDK.

7. Voice Call Control

Several telephony APIs are available on the Pocket PC 2002/2003 for both the Pocket PC and Pocket PC Phone Edition:

- **Phone API:** A high level API used for placing voice calls.
- **Telephony API (TAPI):** A low level API that allows complete control over any available line device.
- **Assisted TAPI:** A single function API used for making a voice call.

The highest-level API, called the **Phone API**, is the easiest and quickest way to add full-featured voice phone capabilities to your application. This API will allow phone calls to be placed and the system call log to be read. The Phone API consists of the following five functions:

- ***PhoneOpenCallLog*:** This function opens the call log and sets the seek pointer to start searching from the beginning of the log.
- ***PhoneSeekCallLog*:** This function initiates a search that ends at a given entry in a call log.
- ***PhoneGetCallLogEntry*:** This function returns the information for the specified call log entry and then advances the seek pointer to the next entry in the call log.
- ***PhoneCloseCallLog*:** This function closes the call log.
- ***PhoneMakeCall*:** This function dials the specified phone number.

The **Telephony API**, or **TAPI**, is by far the most powerful, however is also the most complex, API available for working with telephony devices. The functions are too numerous to list in this document, but a brief overview follows. TAPI provides functions for the following capabilities:

- Retrieve general phone and radio information (serial number, subscriber identity, etc.).
- Retrieve current state of the phone.
- Send and retrieve Unstructured Supplementary Service Data (USSD) messages.
- Deal with caller ID features of the phone.
- Deal with call waiting features of the phone.
- Determine GPRS class of the network.

- Determine the High Speed Circuit Switched Data (HSCSD) configuration.
- Retrieve and manipulate the radio state.
- Establish any supported telephony or radio connection.

A simple but not very functional API is **Assisted TAPI**. Assisted TAPI will only allow a call to be placed. It actually passes on your request to a separate call-management application. This application is then responsible for placing and controlling the call. On the Siemens SX56, this is the built-in phone application. Assisted TAPI consists of the following function:

> ***tapiRequestMakeCall*:** Use this function to request the establishment of a voice call. A separate call-management application is responsible for establishing the voice call on behalf of the requesting application. This voice call is subsequently controlled by the user's call-manager application.

For more detailed information on the APIs discussed above, see Microsoft's developer Web site, http://msdn.microsoft.com/.

8. Short Message Service (SMS) and SIM Access

Some applications may want to send or receive messages using the Short Message Service (SMS) or to access the information in the Subscriber Information Module (SIM).

SMS enables wireless devices to send and receive short messages through an SMS center operated by the cellular operator. A single short message can contain up to 160 alphanumeric characters (or 224 characters if using 5-bit mode). Messages can contain any combination of alphanumeric characters. Non-text (binary) messages are also supported. Some devices store SMS messages on the SIM, whereas others do not. For example, on the SX56, SMS messages are not stored on the SIM card. For more information about SMS, refer to 3GPP Technical Specification 03.40 referenced in Section 1.3.3.

With the Pocket PC Phone Edition, users can send and receive SMS messages using Microsoft Outlook Inbox.

There are several ways you can have your application manage SMS messaging in Pocket PC Phone Edition. The simplest way is to use the Short Message Service (SMS) API. Using this API, you can write applications that send and retrieve SMS messages in relatively few lines of code. The SMS API allows developers to do things like send and receive messages, get the phone number of the sending device and receive SMS notifications. For more information on the SMS API see Microsoft's developer Web site, http://msdn.microsoft.com/.

Note: This API refers to how applications access messages within the local Pocket PC Phone Edition environment.

One important change to the SMS API in Pocket PC 2003 is the deprecation of the SMSReadMessage function. This has been replaced by a new CEMAPI interface: ***IMailRuleClient***. The interface, which must be implemented by the application, allows direct notification of new messages as the device receives them. For information on how SMS messages are delivered to devices from external networks, refer to AT&T Wireless devCentral at http://www.attwireless.com/developer/.

Another closely related API is the Subscriber Identity Module (SIM) Manager API. The SIM Manager API exposes a high level set of functions that allow a developer to interface with the SIM card attached to a device.

There are functions within the API that allow the developer to perform the following operations:

- Determine the SIM card capabilities
- Read SMS messages from a particular storage location.
- Lock and unlock the phone
- Read and write phone book entries on the SIM card
- Get the status of the SIM card storage
- Change the SIM card locking password

SMS messaging and SIM manipulation can also be performed directly using serial communications and the AT command set. This is the most complex way to perform SMS messaging and SIM card interaction. But it does allow the developer to access all of the capabilities of the SIM card. Most of the commands that are available with the various SIM cards are the same, but most phone manufacturers have unique and extended commands. For this reason, you will have to consult the developer documentation for each phone you want to support.

The main reason for performing raw programming to the SIM card would be if you needed to access custom functionality within the card. However, the basic process for accessing the SIM card is fairly straightforward. You first open the serial port using the ***CreateFile*** function. You could then use the serial communications API to send commands and get the status of the port. When dealing with ports, all the standard file programming APIs are available on the Pocket PC device. These include items such as asynchronous I/O and low-level file I/O like ***WriteFile***.

It is also important to keep in mind that other applications, such as Pocket Outlook, have the ability to perform SMS messaging. Applications you write must be sensitive to locking issues related to the SIM card. For instance, if Pocket Outlook is attempting to retrieve SMS messages, your application will be unable to gain a lock on the SIM card. This means you must insert retry and fail logic as appropriate. Since this is also a shared resource, your application should obtain a lock only as long as it is actually required. Specifically, you should not obtain a lock when the application starts and then release it when the application terminates, as this will prevent any other applications from obtaining a lock.

9. Bluetooth Integration

New to Pocket PC 2003 is native Bluetooth integration. This API gives the developer the capability to utilize the Bluetooth radio on Bluetooth equipped devices. The API includes the following functions:

- ***BthGetMode***: This function allows the application to determine the current mode of the Bluetooth radio. These modes are connectable, discoverable, and off.
- ***BthSetMode***: This function allows the application to set the mode of the Bluetooth radio.

Although not directly related to GPRS/EDGE, an application writer could utilize the Bluetooth connection between a PC (or notebook computer) and a Pocket PC 2003 device to create an Internet connection through the mobile device using GPRS/EDGE.

10. Software Development Tools

Microsoft provides a variety of development tools for Windows CE. Different tools are better suited for different scenarios. Microsoft recommends that you use the tools as shown in Table 4.

Table 4: Available Tools and When to Use Them

Tools	When to Use
eMbedded Visual C++ (using eMbedded Visual Tools 3.0 and Pocket PC 2002 SDK)	Native code applications for Pocket PC 2000 or Pocket PC 2002. Drivers for the Pocket PC. Applications with real-time performance requirements. COM servers or Microsoft ActiveX® controls.
eMbedded Visual C++ (using eMbedded Visual C++ 4.0 SP2 and Pocket PC 2003 SDK)	Native code applications for Pocket PC 2003 Drivers for the Pocket PC Applications with real-time performance requirements. COM servers or Microsoft ActiveX® controls.
.NET Compact Framework (using Visual Studio .NET 2003)	Managed code applications for Pocket PC 2000, 2002 and 2003. Pocket PC Applications that use rapid development methodologies or call XML Web services. Applications that must work well in either a connected or disconnected environment. Applications that use either Visual Basic .NET or C#. Using the same tools for desktop, server, and device development. Software that provides a reliable and secure environment.

Note: There is a separate SDK for Windows Mobile-based Smartphone.

For information regarding Visual Studio .NET 2003 and the .NET Compact Framework please visit
http://msdn.microsoft.com/vstudio/device/overview.asp

For an introduction to developing applications for the Pocket PC, and for the latest tools, technical articles and resources, visit: http://www.microsoft.com/windowsmobile/developer. You may also want to consider joining the Microsoft Mobile Solutions Partner Program that provides assistance in this area.

The following sections examine the principal development tools in greater detail.

10.1 eMbedded Visual C++

eMbedded Visual C++ is a standalone application for developing PocketPC software using C++. It has the same look and feel as Visual Studio 6.0, and if you are currently a Visual C++ developer, you will be very familiar with the user interface. The initial version of the C++ sample application was written using eMbedded Visual C++ 3.0. For Pocket PC 2003 development, eMbedded Visual C++ 4.0 must be used. It contains many new features such as an Active Template Library (ATL) out-of-process wizard, a subset of the Standard Template Library (STL), and Run-Time Type Identification (RTTI) just to name a few.

Using eMbedded Visual C++ 3.0 you will be able to target the following platforms:

- Pocket PC and Pocket PC 2002
- Smartphone 2002
- H/PC Pro
- Palm-size PC 1.2

Using eMbedded Visual C++ 4.0 you will be able to target the following platforms:

- Pocket PC 2003 (all editions)
- Smartphone 2003

Both products are available as free downloads from Microsoft's developer Web site (http://www.microsoft.com/windowsmobile/developer). Visit the developer downloads section.

EMbedded Visual C++ is the recommended Microsoft tool for development of native (non-.NET Compact Framework) applications. You also have the ability to run both eMbedded Visual Tools 3.0 and

eMbedded Visual C++ 4.0 SP2 side-by-side if doing development for several Pocket PC versions simultaneously although there are some limitations, for example having two emulators open.

10.2 Windows CE Platform SDK

The Windows CE Platform SDK is designed to provide tools for building generic applications for Windows CE .NET version 4.2. It contains header files, libraries, documentation, samples, and various tools to aid in the development of those applications. However, it does not include the cross-compilers needed for developing binaries that run on Windows Mobile-based Pocket PCs.

The following are some basic requirements for development using the SDK:

- To ensure proper functionality of emulation, you need to run Windows NT 4.0 or greater.
- Microsoft Visual C/C++ or another C/C++ compiler must be installed.
- Windows CE Platform SDK must be installed.

10.2.1 SDK Emulation Environment

There is a new emulator that ships with the Pocket PC 2003 SDK. This emulator now runs as a true hardware virtual machine. This means that the emulator more accurately imitates an actual Pocket PC 2003 device. It also supports networking, which means that debugging applications that use networking functionality is much easier. Fortunately, running and debugging with the emulator is the same as with emulator that shipped with Pocket PC 2002 SDK. However, the new emulator cannot run side-by-side with previous versions of the emulator. For more information on this emulator see http://msdn.microsoft.com/library/en-us/guide_ppc/htm/intro_to_the_ce_emulator_nafc.asp.

10.2.2 SDK Tools

WM_HIBER.EXE is a tool included in the emulation object store for each platform. The WM_HIBER application allows the user to send WM_HIBERNATE messages to either all applications or the application of their choice. The user can launch it by double-clicking its icon in Explorer, or by calling CreateProcess(). This is important for Windows CE devices,

because hibernation is used when the device is in a low memory situation, and therefore must be supported by your application.

REGSVRCE.EXE allows COM server DLLs to be registered. Therefore, if you are developing a COM DLL, this will allow you to register your module and enable you to debug it.

CEREGEDIT from the menu, or PREGEDIT.EXE from the command line, allows you to alter your device registry under emulation. This will greatly ease debugging when your application uses the registry for storage of various settings.

WINDBG (invoked from the start menu as Window Debugger) is a device-independent application for debugging your Windows CE applications. Due to the extreme flexibility that it provides with respect to COM ports, baud rates, device types, etc., it is not possible to ship it preconfigured for your application. Please consult the Help menu in **windbg** for information on how to configure **windbg** for use in your specific circumstances.

A control panel applet is provided that supplies a way for you to select which object store to use. In selecting an object store, you are limited to selecting object stores residing in the wce\emul\<platform> directory. If you are not familiar with object stores, they will be covered later in this document, (Section 11.3).

10.2.3 SDK Sample Applications for Pocket PC

The SDK also provides a variety of sample applications to get you started with programming for the CE. These applications can be divided into the following categories:

- **User-interface applications:** The user interface samples illustrate ways to use the command bar in ***commctrl***, create a control panel applet, illustrate the handwriting recognition APIs, and many others.
- **ActiveSync modules:** ActiveSync is an API that allows applications to synchronize data between a desktop computer and a companion application running on a Pocket PC device. This is perhaps one of the more difficult aspects of development on the Pocket PC. The samples illustrate how to create a generic desktop ActiveSync module. They also contain a skeleton project that will allow you to quickly create your own ActiveSync module and a sample Stock application that illustrates the use of ActiveSync.

- **Serial driver samples**
- **PCMCIA driver samples**
- **Various device driver samples**

10.3 .NET Compact Framework

This section discusses the new .NET Compact Framework and how this affects mobile application development.

10.3.1 Overview

As part of Microsoft's ongoing .NET strategy, Microsoft has introduced a version of .NET that targets the embedded world. This version is called the .NET Compact Framework. It is essentially identical to the .NET framework except that it has been scaled back to fit on Pocket PC devices. The .NET Compact Framework 1.0 is now included in Pocket PC 2003 ROM. This simplifies distribution of applications that are targeting Pocket PC 2003. The following list details the main characteristics of .NET:

- **Interpreted Code:** Code that is compiled to target .NET; it is not machine code, but is interpreted like Java byte code. Microsoft refers to this as an Intermediate Language (IL).
- **New Language:** C# has been introduced as a language to take full advantage of the .NET framework and make it easier to program with .NET.
- **Logical Class Hierarchy:** Instead of a flat, often ambiguous API like Win32, the .NET framework consists of many logically designed and arranged classes. Many of these classes are simply wrappers that use the Win32 API, but make it easier and more efficient.

One of the main advantages of .NET applications is that they are not compiled to native code. This means that the code that has to be deployed is the same, regardless of the processor of the device. Since the IL is identical regardless of the platform, the same binaries can be distributed. However, this same benefit could be an issue if there is not a .NET Compact Framework class available for the function you want to perform. Such is the case with the .NET sample application later in this document. For example, the .NET Compact Framework does not have any classes available that expose the connection manager API, therefore these must be accessed using the .NET interop services.

Also new to .NET is a new language called C# (pronounced see-sharp). This language is a freeform C-style language. In fact, C++ programmers will have little trouble becoming effective C# programmers, due to the similarities between the two languages. However, when developing .NET applications, you will find using C# to be much easier and more efficient than C++, since C# was designed with .NET in mind.

The new class hierarchy exposed via .NET also simplifies development. Unlike the Win32 API, which is flat and non-object oriented, the .NET classes are ordered into specific hierarchical namespaces. This allows you to easily find and understand the classes that your application might require. And since all the .NET functionality is grouped into classes, it greatly simplifies the programming model as well.

Microsoft has recently released the .NET Compact Framework technology, a platform and runtime that enable enterprise developers to take advantage of a single Visual Studio .NET integrated development experience for managed code and XML Web services on mobile devices.

The Visual Studio.NET 2003 is currently available to MSDN Universal subscribers from MSDN Subscriber Downloads at: http://msdn.microsoft.com/subscriptions/resources/subdwnld.asp or can be purchased independently.

It is important to note that Visual Studio .NET 2003 creates binaries that target the .NET Compact Framework. Only applications targeting the .NET Compact Framework can be created for the Pocket PC platform using this tool. To create native applications, you must use eMbedded Visual C++ 4.0.

10.3.2 Emulation

Also part of .NET development is a new emulation environment. As opposed to the previous emulator that used the Win32 APIs to draw and execute the emulator, the new emulator runs the exact Pocket PC code from within a virtual machine. It is a powerful high-fidelity device emulator that is integrated within the Visual Studio .NET design environment. This emulator accurately represents the physical device by executing the exact target operating system on the developer's desktop, which runs actual operating system images in a virtual machine. Emulation operating systems for Pocket PC devices are provided. In fact, as new custom devices based on Windows CE .NET emerge, their emulators can easily

be plugged into Visual Studio .NET to speed application development for those devices.

10.3.3 Sample Applications

As with all other Microsoft development tools, there are plenty of sample applications available with the .NET compact framework tools. Sample applications are currently available demonstrating tasks and techniques in the following areas:

- Creating WinForms applications
- Authoring components
- Creating custom controls
- Structured exception handling
- Calling and creating XML Web services
- Programming SQL Server CE databases

These sample applications are all available from Microsoft's developer Web site: http://msdn.microsoft.com.

11. Programming Considerations

This section discusses programming considerations, including the characteristics of wireless connections and some unique aspects of the Pocket PC environment, including memory management, object store, the property database, and UDP/TCP/IP sockets communications.

11.1 Characteristics of Wireless Connections

Common for all cellular-data networks, applications must take into account data-throughput rates, latency, and connection characteristics. GPRS offers typical throughput rates of 30 to 40 Kbps (depending on type of device) and EDGE offers typical throughput rates of 110 to 130 Kbps. The Siemens SX-56, for example, is a GPRS class 10 device that allows downlink speeds using four time slots (40 Kbps effective maximum throughput) and uplink speeds using two time slots (20 Kbps maximum). These data rates support a wide range of applications. Developers should also take pricing plans into account.

Note: Effective maximum throughput means the highest throughput rate delivered to the actual application. Rates are sometimes quoted higher, but include GPRS protocol overhead.

GPRS/EDGE networks have typical latency (or round trip time) of 600 milliseconds, as measured by ping using default (small) packet sizes. This latency will affect the design of the application but not how a program is written. For the end user delays can be noticeable, but if the application is designed to minimize back and forth traffic, delays will not be excessive. This is especially true for content tailored specifically to mobile devices.

It is important to understand the difference between latency and bandwidth. Even though it is possible to achieve connection speeds similar to a landline modem, latency is higher. This causes delays for data items being accessed. For instance, an image may take a moment to begin loading, but will load quickly thereafter. There is little a developer can do to address this delay.

Another aspect of wireless connections, especially in mobile environments, is that connections can be lost in out-of-coverage situations. Though an application cannot prevent the loss of a connection, it can deal with the results of a connection loss, or poor connection. One approach is to cache data during a particular session so that if the

connection fails, that piece of data does not have to be retrieved again. This is particularly important for larger files and data. For example, if an application is receiving a fairly large file or image, it should use a mechanism to retrieve portions of the file at a time. If the connection fails during the retrieval, the application should be able to pick up where it left off. For instance, HTTP chunked data transfer (see Internet RFC 2616) can be used to retrieve pieces of the requested file. Once all the pieces are retrieved, they can be reassembled into the requested file. As part of the architecture for handling poor connections, applications should also be able to handle connections that are dropped.

In the case of a dropped connection, there are a couple of actions an application can choose. First the application could attempt to automatically re-establish the connection, and then give up after a certain time period if a new connection could not be established. For system configurations that support the Microsoft Connection Manager API, this could be done using the ***ConnMgrEstablishConnection*** function.

Alternately, the application could prompt the user and only attempt to re-establish the connection based on the user input. Unfortunately, an application will not actually be notified that a connection has been lost. It is up to the application to determine if the connection exists or has been broken. Fortunately, this is simple enough to implement. If using sockets, a call to connect would fail, indicating the physical connection has been broken. Also the ***ConnMgrConnectionStatus*** function can be called on system configurations that support the Microsoft Connection Manager API to determine if the physical connection is still available. Ultimately, a scheme should be used that checks the connection for failure after each failed network operation. If the connection has failed, the application could then attempt to reestablish the connection or prompt the user.

11.2 Memory Management

Proper memory management on the Pocket PC is extremely important due to the relatively small amount of memory available. Applications must always check that memory allocations succeeded and must be able to free up memory not currently in use.

The memory architecture of Pocket PC is very similar to the desktop versions of Windows. It supports a virtual address space in which pages are mapped into physical memory. However, on the Pocket PC the virtual address space is 2GB instead of 4GB. Also, when memory is allocated it is not allocated from the paging file, but from physical memory. Page level

memory access is available via routines such as ***VirtualAlloc*** and ***VirtualFree***. However, care must be taken using these routines, since the page size differs between various devices. Typically, an application will be written using the higher-level APIs such as ***alloc*** or ***free***.

Although allocating and freeing memory is largely the same on the Pocket PC as the desktop versions, careful consideration should be taken to optimize memory usage. One cause of poor memory management is fragmentation. As memory is allocated and freed the memory becomes fragmented due to contiguous blocks not being available in a large enough amount. Imagine a scenario where you allocate 10 KB, and then you allocate 5 KB, and then you allocate 10 KB. If you now free the 5 KB of memory, you will only be able to reuse that slot if your allocation is for 5 KB or less. Therefore if you allocate 10 KB, you will now be consuming 35 KB instead of 30 KB. Over time, these fragmentations can add up to become a significant portion of your memory usage.

One way to prevent this is by using the heap allocation routines to manage objects of the same size. Therefore, if your application always allocates objects that are 10 KB and 5 KB, you could create two heaps using the ***HeapCreate*** function. Anytime you needed a new 10 KB object, you could allocate it from the first heap using ***HeapAlloc***. Whenever you needed a new 5 KB object you could allocate it from the second heap. In this scenario you would be guaranteed not to fragment the memory. You should typically only use this scenario when you are allocating a lot of memory blocks of the same size or when all of your de-allocations occur at once.

Memory status is much more important when programming for Pocket PC versus a Windows desktop. If an application needs to allocate a large amount of memory, it should determine first if the memory it requires is available. This can be done using the ***GlobalMemoryStatus*** function. The following sample application demonstrates its usage:

```
MEMORYSTATUS memstat;
GlobalMemoryStatus (&memstat);

_stprintf(_T(“Physical memory size: %d KB\n”), memstat.dwTotalPhys/1024);
_stprintf(_T( “Physical memory free: %d KB\n"), memstat.dwAvailPhys/1024);
_stprintf(_T( “Paging file size:     %d KB\n"), memstat.dwTotalPageFile/1024);
_stprintf(_T(“Paging file free:     %d KB\n"), memstat.dwAvailPageFile/1024);
_stprintf(_T(“Virtual memory size:  %d KB\n"), memstat.dwTotalVirtual/1024);
_stprintf(_T(“Virtual memory free:  %d KB\n"), memstat.dwAvailVirtual/1024);
```

In the above example, the printout would typically be something like:

```
Physical memory size: 8192 KB
Physical memory free: 5324 KB
Paging file size:     0 KB
Paging file free:     0 KB
Virtual memory size:  32768 KB
Virtual memory free:  30152 KB
```

This paging file information will always be 0 KB since there is no paging file. The virtual memory size will always be 32 MB since that is the amount allocated to each process. In this example, the device has 8 MB of physical memory. When running under emulation, 16 MB will always be returned.

Using the information returned, you could programmatically take a different course of action if there is not enough memory available to perform an operation. You could free memory that is not currently being used, or you could prompt the user to attempt to close other running applications that may be consuming physical memory.

Three critical low memory events will typically take place in the following sequence:

1. **WM_HIBERNATE**: This is the first of the low memory events that occur. Applications should be ready to respond to this message. The shell will send this message to the application that has been inactive the longest, when memory is low (depending on device page size) and a new allocation is attempted. An application should free up any unused resources at this time and close any unnecessary windows.
2. **WM_CLOSE**: This is the next of the low memory events. This message will be sent when the device reaches a low memory threshold. When an application receives this notification, it should save any open documents without prompting the user and free all resources.
3. **Critical Memory Event**: This is the last and most severe. At this point the CE will not allow any new processes to be created.

11.3 Object Store

Pocket PC uses the Object Store for storing files, databases, and the registry. The Object Store uses RAM for its storage. Unlike the desktop versions of Windows, Pocket PC does not use drive letters for devices. It uses a scheme very similar to that used in Unix variants. Devices are

connected to the file system via folders in the Object Store. Just like on a Unix system where you may access your CD-ROM drive by mounting it to ***/mnt/cdrom***, and then changing to that directory to read the contents, you may have a storage card mounted to ***\Storage Card*** and be able to read and write to that directory to read and write to the storage card.

On most Pocket PC devices, storage devices are typically removable. Therefore, applications that require storage should be ready to respond to the ***WM_DEVICECHANGE*** message. This message is sent whenever a device is attached or removed. One exception to this rule is when the Pocket PC is powered on. Pocket PC will send two ***WM_DEVICECHANGE*** messages to indicate a device is being removed and then inserted.

The following code snippet shows how you could capture the message and perform some action based on whether a device was getting removed or inserted:

```
case WM_DEVICECHANGE:
  switch (wParam)
  {
  case DBT_DEVICEARRIVAL:
    OnDeviceAdd ((DEV_BROADCATS_HDR*)lParam);
    break;
  case DBT_DEVICEREMOVECOMPLETE:
    OnDeviceRemove ((DEV_BROADCAST_HDR*)lParam);
    break;
  }
  break;
```

In the sample above, had you prompted the user to insert their media device, you could then respond to that event in the ***OnDeviceAdd*** function, and continue processing for the user.

11.4 Property Database

The property database in Pocket PC is a database that allows applications to store structured data. The property database uses the Object Store for its storage, and stores the databases in a folder called database. You will find the standard databases in this folder, such as appointments, contacts, and tasks databases.

Just like files and folders, each property in the database has a unique Object ID (OID). This OID is used when accessing the property or record. The Pocket PC API allows full access to properties with the ability to

create, destroy, access, sort, and search a property database. The following example shows how you would create your own property database to store application relative data.

```
CEGUID ceguid;

if (CeMountDBVol (&ceguid, _T("\\Storage Card\\MyVolume.CDB"), CREATE_NEW))
{
  // TODO: Add code here to manipulate your new property database
}
else
{
  _stprintf(_T(“Error: could not create new property database.\n"));
}
```

In the example above, there are a couple items to notice. The ***CeMountDBVol*** function is used to create this new database. This is similar to how one might use the ***OpenFile*** function to create a new file. ***CeMountDBVol*** can also be used with other flags such as ***CREATE_ALWAYS***, ***TRUNCATE_EXISTING***, and so on. Another item to notice is the path to the new database, \Storage Card\MyVolume.CDB. This path is the root directory on an attached storage card.

Another interesting aspect of items in the property database is that up to four sort orders can be associated with properties to speed up searching for records. The following example shows how you might specify a sort order for a property database containing stock tickers.

```
CEDBASEINFO cedbinfo;
cedbinfo.dwFlags = CEDB_VALIDSORTSPEC;
cedbinfo.wNumSortOrder = 1;
cedbinfo.rgSortSpecs[0].propid = MAKELONG (CEVT_LPWSTR, 100);
cedbinfo.rgSortSpecs[0].dwFlags = CEDB_SORT_CASEINSENSITIVE;

// Assumes ceguid has already been initialized with a call
// to CeMountDBVol. 100 is the OID of our particular database
//
CeSetDatabaseInfoEx (&ceguid, 100, &cedbinfo);
```

In the example above, you can see it is fairly simple to change the sort order of one of the properties to be case insensitive. If the database already has records, this process can be very time consuming; therefore, you should prompt the user if necessary.

11.5 UDP/TCP/IP Sockets Programming

If you are familiar with sockets programming, you will be familiar with sockets programming on the Pocket PC platform. This paper assumes you are already familiar with IP, TCP, and UDP fundamentals. The sockets library included with Pocket PC is a subset of the sockets library available on the desktop versions of Windows. The most notable functionality missing from the sockets library is asynchronous support. This means that functions such as ***WSAAsyncSelect*** are not supported. However, since threads are available, it is possible to create a worker thread to perform communications and still keep the user interface responsive.

When writing a sockets application for Pocket PC, the first thing to do is initialize the sockets library. Calling the ***WSAStartup*** function does this. The current version supported is 1.1; therefore, a code snippet to initialize the sockets library would be:

```
WSADATA wsadata;
if(0 == WSAStartup (MAKEWORD(1,1), &wsadata))
{
  // Winsock has been initialized, do your comm. now
}
else
{
  // Winsock could not be initialized
}
```

Like most sockets functions, ***WSAStartup*** returns zero when successful, and a non-zero value upon failure. To determine the reason for failure, call ***WSAGetLastError***. In order for sockets to be used, the Pocket PC must have some sort of networking device attached. For the Siemens SX56, sockets can be used via GPRS/EDGE with the built in cellular modem. However, a GPRS/EDGE connection must first be established.

12. Sample Application

The following sections show the implementation of the stock-quote sample application using first the eMbedded Visual C++ environment, followed by the Windows CE .NET environment.

12.1 eMbedded Visual C++

Development using eMbedded Visual C++ closely resembles application development using Visual C++ for desktop applications. The main differences are in building and debugging, and not necessarily the code itself. The code in the sample application could actually be used with little modification to create a similar application for the desktop. This means that if you are already a Visual C++ developer, you will already be comfortable with the development environment and will quickly be productive writing code.

12.1.1 Set Up the Environment

You will need to configure your environment as follows:

1. Windows NT/2000/XP.
2. The latest service pack. This will vary depending on the OS you are using for development. Check the latest installation requirements for eMbedded Visual C++ 4.0 SP2.
3. Microsoft eMbedded Visual C++ 4.0 SP2.
4. Once you have your environment set up you will be ready to begin creating the sample application.

12.1.2 Creating the Project

At this point, you should launch the eMbedded Visual C++ 4.0 Integrated Development Environment (IDE). Once you have the IDE running, you will notice that it is the same as the Visual C++ 6.0 IDE. Now that you have it running, follow the steps below to get the base project created:

1. Select **New…** from the **File** menu.
2. On the **New Projects** dialog, select **WCE Pocket PC 2003 MFC Appwizard (exe)** project type. Since we will be targeting the

Siemens SX56 for our sample application we will be able to create an MFC application for the Pocket PC.

3. In the **Project Name** field, type in StockQuote as the name of the project.
4. Verify the directory the project will be created in is acceptable.
5. Ensure that at least **Win32 (WCE ARM)** and **Win32 (WCE x86em)** are selected in the **Projects** list.
6. Click **OK**.
7. You should now see the **Step 1 of 4** dialog. At this point select **Dialog Based** as the type of application.
8. Click **Next**.
9. You should now see the **Step 2of 4** dialog. Check the **Windows Sockets** checkbox.
10. Click **Next**.
11. You should now see the **Step 3 of 4** dialog. Click **Next**.
12. You should now see the **Step 4 of 4** dialog. Click **Finish**.
13. You will now see a project summary dialog. Click **OK**.

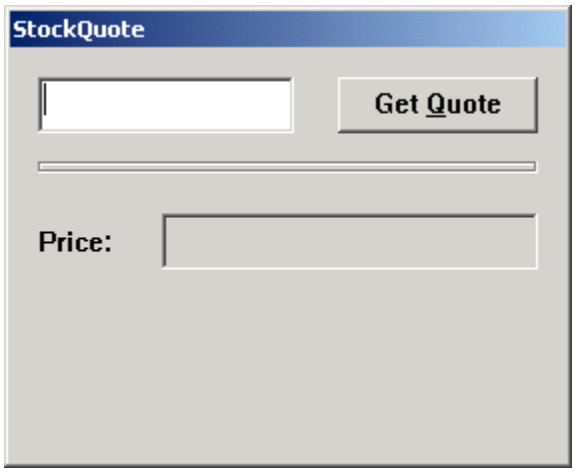
At this point, the base project is created. You will notice during the project creation you selected **Windows Sockets** support and created the project to build with MFC as a shared DLL. The first option simply inserts a **#include <afxsock.h>** in the **stdafx.h** file. This file, **afxsock.h**, essentially just includes **winsock.h**, and defines some helper classes that can be used to work with sockets.

Including MFC support as a shared DLL can be a more difficult decision. If you are using MFC, you can choose to include it as a DLL or a shared library. The advantage of including it as a DLL is that your application will have a smaller footprint. However, you will have to be certain that the variant of the OS supports MFC and has the **mfcce300.dll** available or you will have to include it with your application distribution. If using MFC as a static library, you will not have to worry about the presence of the mfcce300.dll, but your application will have a much larger footprint. These factors should be considered when developing and before deploying your application. In the case of the Siemens SX56, the DLL is already included; therefore, the shared DLL option was chosen to reduce the size of the application.

12.1.3 Writing the Code

1. First, you need to update the main dialog.
   a. Click on the **Resource** tab, expand the **Dialog** folder, and open the IDD_STOCKQUOTE_DIALOG dialog.
   b. Three important items will be added to the dialog. The edit box for the user to enter the ticker symbol, a button to obtain the quote, and a text box to display the current quote. Add these items to the dialog now with the following names: IDC_TICKER, IDC_GETQUOTE, IDC_PRICE. Due to the form factor of the Siemens SX56 display, the dialog is kept minimal. Unfortunately, using the resource editor in this manner does not allow the targeting of various form factors. You would have to write specific code to resize and move windows based on screen dimensions if you wanted your application to be extremely portable.

Figure 1: Stock Quote Dialog Box



2. Now you will add some objects to our main dialog class to represent each of the controls on the dialog. Open the file **StockQuoteDlg.h** and edit it:

```
class CStockQuoteDlg : public Cdialog
{
public:

        CStockQuoteDlg(CWnd* pParent = NULL);

        //{{AFX_DATA(CStockQuoteDlg)
        enum { IDD = IDD_STOCKQUOTE_DIALOG };
        CEdit m_wndPrice;
        CEdit m_wndTicker;
```

```
        CButton m_wndGetQuote;
        //}}AFX_DATA

        //{{AFX_VIRTUAL(CStockQuoteDlg)
        public:
        virtual BOOL DestroyWindow();
        virtual BOOL PreTranslateMessage(MSG* pMsg);
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);
        //}}AFX_VIRTUAL

protected:

        HICON m_hIcon;

        //{{AFX_MSG(CStockQuoteDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnGetquote();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()

private:

        bool m_bHaveWinsock;
};
```

You will notice a few things have been added. There are two CEdit objects: One for the ticker and one for the price. There is a CButton object added for the **Get Quote** button. Overrides for DestroyWindow and PreTranslateMessage have been added. Their implementations will be explained later. Finally a message handler, **OnGetQuote**, has been added, which is called when the **Get Quote** button has been pressed. If you are familiar with MFC programming, you will notice this code is no different than what you might write for a desktop application. Many functions and styles are not available, but MFC for the Pocket PC is fairly complete. Most of the limitations actually stem from the underlying Windows API used by MFC.

3. Next, the window objects need to be hooked up. This can be done by updating the default DoDataExchange implementation, which was created by the wizard. By calling the DDX_Control function, you can have MFC automatically wire a particular class variable to a window. So update the function now to appear as follows:

```
void CStockQuoteDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CStockQuoteDlg)
        DDX_Control (pDX, IDC_PRICE, m_wndPrice);
        DDX_Control (pDX, IDC_TICKER, m_wndTicker);
```

```
        DDX_Control (pDX, IDC_GETQUOTE, m_wndGetQuote);
        //}}AFX_DATA_MAP
}
```

As you can see, the program simply passes in the ID of the control and a reference to the object to be associated with that control. This function will be called by the framework when the dialog is created. It is also called when **UpdateData** is called to update windows with the value of variables or vice versa.

4. Moving down through the file, you can now update the message map. MFC uses several macros to enable a function call to be tied to a windows message. You can simply update the message map so that when the user presses the **Get Quote** button, the **OnGetQuote** function will be called. So update the message map as follows:

```
BEGIN_MESSAGE_MAP(CStockQuoteDlg, CDialog)
        //{{AFX_MSG_MAP(CStockQuoteDlg)
        ON_BN_CLICKED(IDC_GETQUOTE, OnGetquote)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

5. Now you need to update the OnInitDialog function that was created with code to initialize the sockets library. You can call the function, **WSAStartup**, to attempt this. The important parameter to pass is the version of Winsock you need loaded. At the time of this writing, version 1.1 is supported by the Pocket PC. So now update the **OnInitDialog** function to appear as follows:

```
BOOL CStockQuoteDlg::OnInitDialog()
{
        // This will be set to false if we can initialize Winsock
        // This will allow us to set focus to the ticker edit window
        //
        bool bRet = true;

        CDialog::OnInitDialog();

        SetIcon(m_hIcon, TRUE);             // Set big icon
        SetIcon(m_hIcon, FALSE);            // Set small icon
        CenterWindow(GetDesktopWindow());
        // center to the hpc screen

        // Attempt to start up Winsock 1.1
        //      WSADATA wsad;
        if (0 == WSAStartup (0x0101, &wsad))
        {
                m_bHaveWinsock = true;
```

```
                m_wndTicker.SetFocus ();
                bRet = false;
        }
        else
        {
                // Notify the user and disable some of our controls
                //
                MessageBox (_T("Unable to start up Winsock.\n"Please ensure you
have a network device
                                Connected and enabled."),
                                _T("StockQuote Error"),
                                MB_OK | MB_ICONERROR);

                m_bHaveWinsock = false;

                m_wndTicker.EnableWindow (FALSE);
                m_wndGetQuote.EnableWindow (FALSE);
        }

        // return true unless we were able to init Winsock &
        // set focus to the ticker window
        //
        return bRet;
}
```

6. As you may have noticed, the program added an override for the **PreTranslateMessage** function. The reason for this is that the application is a dialog based application and uses the default dialog box procedure. The problem is, when the user presses the **Enter** key, the default action is to close the dialog. On the Siemens SX56, the **Enter** key can be activated using the touch screen keyboard or via the center button below the screen. Unfortunately, it is instinctive to press the **Enter** key after typing in a ticker symbol. Therefore, the program needs to override the **PreTranslateMessage** function to catch instances when the user has pressed the **Enter** key, and convert that message to appear as though the user pressed the **GetQuote** button. So, update the **PreTranslateMessage** function to appear as follows:

```
BOOL CStockQuoteDlg::PreTranslateMessage(MSG* pMsg)
{
  // This is here to make usage a little easier.
  // - Check to see if the 'Enter' key was pressed
  // - If so, and the focus is in the ticker window
  // - Treat as though the 'GetQuote' button was pressed
  //
  if (pMsg->message == WM_KEYDOWN && pMsg->wParam == VK_RETURN)
  {
        if (GetFocus ()->m_hWnd == m_wndTicker.m_hWnd)
        {
          PostMessage (WM_COMMAND, BN_CLICKED << 16 | IDC_GETQUOTE,
         (LPARAM)m_wndGetQuote.m_hWnd);
```

```
          return TRUE;
        }
  }

  return CDialog::PreTranslateMessage(pMsg);
}
```

7. At this point, you can skip ahead to see how the quote is actually retrieved. All of the work to deal with retrieving the quote has been wrapped into a class called **CQuoteInfo**. This class has two public functions and a single helper function. The first function, **GetPrice**, is used to actually retrieve the current price based on a passed-in ticker symbol. The next function, **Init**, exists to allow the IP address of the stock quote service to be looked up just once, when the user first requests a quote, and then stored for use in subsequent quotes. This increases application performance and decreases network bandwidth usage, which is extremely important in mobile applications. The final function, **FormatRequest**, will create a basic HTTP request string based on the ticker symbol. **GetPrice** only ever calls this function. The following is the code for the **GetPrice** function:

```
bool CQuoteInfo::GetPrice (const CString& cstrTicker, float& flPrice)
{
        bool bRet = false;

        if (Init ())
        {
                SOCKET sockNew = socket (AF_INET, SOCK_STREAM, 0);

                if (sockNew != INVALID_SOCKET)
                {
                        sockaddr_in sai = {0};
                        sai.sin_addr = m_addrQuoteService;
                        sai.sin_family = AF_INET;
                        sai.sin_port = htons (80);

                        if (0 == connect (sockNew, (sockaddr*)&sai, sizeof (sai)))
                        {
                                CString cstrReq;
                                FormatRequest (cstrTicker, cstrReq);

                                // Unfortunately, we need to send as ASCII, not
UNICODE
                                char szReq[1024];
                                int nSendLen = WideCharToMultiByte (CP_ACP,
                                        0, cstrReq, min (sizeof(szReq)-1,
cstrReq.GetLength
                                        ()), szReq, sizeof (szReq), NULL,
NULL);
```

```
                                        if (send (sockNew, szReq, nSendLen, 0) ==
nSendLen)
                                        {
                                                // Get the response at this point
                                                CString strResponse;
                                                char szRecvBuf[1024];
                                                while (1)
                                                {
                                                        int nRecv = recv (sockNew,
szRecvBuf, sizeof
                                    (szRecvBuf), 0);
                                                        if (nRecv == 0 || nRecv ==
SOCKET_ERROR)
                                                        {
                                                                break;
                                                        }

                                                        wchar_t wszRecvBuf[1024];
                                                        MultiByteToWideChar (CP_ACP,
0, szRecvBuf, nRecv,
                                    wszRecvBuf, sizeof (wszRecvBuf));
                                                        strResponse += wszRecvBuf;
                                                }

                                                if (strResponse.GetLength ())
                                                {
                                                        // Look for our symbol quoted
                                                        CString cstrSym = _T("\"");
                                                        cstrSym += cstrTicker;
                                                        cstrSym += _T("\",");

                                                        int nFind = strResponse.Find
(cstrSym, 0);
                                                        if (nFind != -1)
                                                        {
                                                                nFind +=
cstrSym.GetLength ();
                                                                swscanf
((LPCTSTR)strResponse + nFind, _T("%f"),
                                                &flPrice);
                                                                bRet = true;
                                                        }
                                                }
                                        }
                                }

                        closesocket (sockNew);
                }
        }

        return bRet;}
```

The first thing this function does is call the **Init** function. **Init** will perform a DNS lookup on the server being used for the quote service and will then cache the IP address. If **Init** returns successfully, the program will now create a new socket with a call

to the **socket** function. The program will then call the **connect** function to connect to the stock quote server.

Note: Many socket functions return zero when they are successful. So, be cautious when analyzing the return codes as this is a large source of defects when programming with Winsock.

It is also important to note that for a wireless device, **connect** will not actually establish a link via a GPRS/EDGE or other networking device. It will only attempt to establish the TCP link over an existing route. Therefore, if your application is responsible for acquiring a link, the link should be established before your socket calls are made.

8. The program can then call **FormatRequest** to create the HTTP request string and then send the request to the server using **send**. You will notice that **send** returns a positive value equal to the number of bytes sent when it is successful. The program will then retrieve the response using the **recv** function. This function will return zero when the server has closed the connection. The program will simply loop through receiving data until the connection has been terminated. Ideally, since this is a mobile application, you would like to keep the connection open as long as the application is open to minimize network bandwidth usage. Unfortunately, the quote server does not support keeping the connection open indefinitely. If you had the opportunity to design all aspects of this system, this would be an important consideration when developing the server application.
9. Finally, the program will attempt to extract the price from the return string. As you can see, the program looks for the ticker symbol in the return string, and then extracts the next comma-separated value. Ideally, the program would parse the HTTP headers and body to extract all the information needed, but for the sake of brevity and simplicity in this sample application, the program will extract exactly what is needed from the return string.

12.1.4 Connection Manager

As stated in the previous section, the program must have a connection over some networking device before the program can create a sockets connection. In order to achieve this, there is a new set of functions available for Pocket PC 2002/2003: the Connection Manager API. These functions abstract all of the underlying details of obtaining the connection. In the situation where multiple network devices are available, the module

will automatically determine the best device to use based on the route to the resource requested.

For complete documentation on the Connection Manager API, see the Pocket PC 2002/2003 SDK (Referenced in Section 1.3.2). In the sample application, you will use just a few of the functions available to create the connection: **ConnMgrApiReadyEvent**, **ConnMgrMapURL**, **ConnMgrEstablishConnectionSync**, **ConnMgrConnectionStatus**, and **ConnMgrReleaseConnection**. **ConnMgrApiReadyEvent** will be called to ensure that the connection manager subsystem is started and ready to accept requests. The **ConnMgrMapURL** function is used to get a Global Unique Identifier (GUID) that represents the network to use for the resource you are attempting to access. Basically, you call the function with the URL or resource you are going to access, and the function will return a GUID for use with the **ConnMgrEstablishConnctionSync** function.

At this point, **ConnMgrEstablishConnectionSync** is called to actually create the connection. If it returns successfully, you are now free to use sockets or other higher level APIs to access your network resource. In the case of the sample application, the program uses the **connect**, **send**, and **recv** functions at this point to get the stock quote. It is also important to note that the Connection Manager API functions return HRESULT types. Therefore you should always use the **SUCCEEDED** or **FAILED** macros to determine the result of a function, unless you are looking for a certain result code. Comparing with values such as **S_OK** or **E_FAIL** will ultimately lead to programming errors.

```
class CconnMgr
{
public:
        CConnMgr ()
        {
                m_hConnection = INVALID_HANDLE_VALUE;
                m_bFailed = false;
                m_bStarted = false;
        }

        ~CConnMgr ()
        {
                if (m_hConnection != INVALID_HANDLE_VALUE)
                {
                        ConnMgrReleaseConnection (m_hConnection, true);
                        CloseHandle (m_hConnection);
                }
        }

        bool IsConnected ()
```

```
	{
		bool bRet = false;
		if (m_hConnection != INVALID_HANDLE_VALUE)
		{
			DWORD dwStatus;
			if (SUCCEEDED(ConnMgrConnectionStatus (m_hConnection,
&dwStatus)))
			{
				bRet = CONNMGR_STATUS_CONNECTED & dwStatus ? true :
false;
			}
		}
		return bRet;
	}

	bool Start ()
	{
		if (!m_bStarted)
		{
			HANDLE hWait = ConnMgrApiReadyEvent();
			if (hWait != INVALID_HANDLE_VALUE)
			{
				if (WAIT_OBJECT_0 == WaitForSingleObject (hWait,
60*1000))
				{
					m_bStarted = true;
				}
				CloseHandle (hWait);
			}
		}

		return m_bStarted;
	}
	bool Connect (HWND hwnd)
	{
		bool bRet = false;
		if (Start () && !m_bFailed && !IsConnected ())
		{
			// Find the guid for our network
			CONNMGR_CONNECTIONINFO ci = {0};
			TCHAR szUrl[] = _T("http://quote.yahoo.com/");
			DWORD dwIndex = 0;
			if (SUCCEEDED (ConnMgrMapURL (szUrl, &ci.guidDestNet,
&dwIndex)))
			{
				// Attempt to connect us
				ci.cbSize = sizeof (ci);
				ci.dwParams = CONNMGR_PARAM_GUIDDESTNET;
				ci.dwPriority = CONNMGR_PRIORITY_USERINTERACTIVE;
				ci.dwFlags = CONNMGR_FLAG_PROXY_HTTP;
				ci.bExclusive = false;
				ci.bDisabled = false;
				ci.lParam = (LPARAM)0;
				DWORD dwStatus = 0;
				HRESULT hrRet = S_OK;
				if (SUCCEEDED(hrRet = ConnMgrEstablishConnectionSync
(&ci, &m_hConnection, 30*1000, &dwStatus)))
				{
```

```
                                        bRet = dwStatus & CONNMGR_STATUS_CONNECTED ?
true : false;
                                }
                                else
                                {
                                        CString cstr;
                                        cstr.Format (_T("ConnMgr fail - %08lX"),
hrRet);
                                        MessageBox (hwnd, cstr, _T("ConnMgr error"),
MB_OK);
                                }
                        }
                        else
                        {
                                MessageBox (hwnd, _T("Could not map URL"), _T("Map
URL Error"), MB_OK);
                        }
                }
                if (bRet == false) m_bFailed = true;
                return bRet;
        }private:       HANDLE m_hConnection;  bool m_bFailed;
        bool m_bStarted;};
```

As you can see from the code snippet above, all of the connection manager functionality that we use in the sample application has been wrapped into a clean and easy to use class. At this point, the class can be used by our main dialog class to ensure that a connection exists before attempting to get a stock quote.

12.1.5 Debugging

Debugging an application on a Pocket PC is similar to application debugging on a PC. The first rule is to always attempt to perform the debugging using the emulator. The main reason for this is speed; debugging on the emulator is significantly faster than debugging on the real device. You can easily set breakpoints and single step through your code. However it is not always possible to do this, especially when the error only occurs on the real device.

One area where you will encounter this problem is working with network connections. You will notice when your PC is running in the emulator, as long as your PC has a network connection, the application will always perform correctly and fetch the quote quickly. However, when attempting to run the application, on an actual device like the SX56, you may get the message Unable to retrieve quote. This means you will have to perform debugging on the device to see where it is failing.

Debugging on the device is similar, except that it is significantly slower when single stepping the code. This is due to the amount of

communication occurring between the CE device and the PC over the USB cable. Because of this, strategies should be built into your code to minimize the amount of single step debugging that needs to take place. First, always be specific about the error condition that caused failure and report this in the user interface. This may allow you to pinpoint the error without having to debug at all. This may also mean you need to debug specific handling since you may not be allowed to display this level of error to the user.

Another approach is to simply use debug printing to display the error. This can be done with the **OutputDebugString** function. Messages printed using this function will be caught and displayed by the connected debugger. When you are in the eMbedded Visual C++ environment, you simply have to select Build→Start Debug→ Go. Then make sure the output window is visible. Now you can run the application through the steps used to produce the error, and you will see your debug messages printed to the output window of the development environment.

This approach works well for isolating the location of the error, since the application runs at nearly full speed when you are not single stepping through the code. It is also advisable to not print out debug messages in production code, unless you are targeting a developer program for your production build, such as a beta program. For this reason, you should use a wrapper such as the following to allow debug printing to be compiled out of release builds:

```
#ifdef _DEBUG
#define OUTDEBUGMSG(x) \
{\
  OutputDebugString (x);\
}
#else

#define OUTDEBUGMSG(x) {}
#endif
```

12.1.6 Summary

At this point the application is complete. Not all of the source code has been listed in the body of this document, so you should extract the code from the AT&T Wireless developer Web site and then build it. See http://www.attwireless.com/developer/technologies/pocketPC/

For simplicity, the sample code does not attempt to reestablish a lost connection, but your applications should attempt to do so. For your first build, you should use the x86em debug build. This will build the executable for the emulation environment. You will then be able to run and debug the application just like any other Windows application. You can set breakpoints in your code and step through it just as you would with Visual C++. Once you have run the code and debugged it, you should then build it for the target device. For the Siemens SX56 case this is the ARM release build. Once you built it, you can upload it to the device using the ActiveSync program. The default project settings will attempt to do this for you when you build a non-emulation target. By default, it will place your new executable in the **\Windows\Start Menu** folder, so it will appear on the start menu of the device.

You should now have a good understanding of how to develop, build, and debug an application using eMbedded Visual C++ 4.0 SP2. If you have developed using Visual C++ before, then most of the environment should be very familiar to you. You should also note that most of the code is identical to what would be written for a desktop application. Some of the important differences are due to limitations in the Windows API, such as Unicode-only support. This means that if your project requires communication to standard services, such as HTTP or FTP, you will likely be performing many conversions to and from Unicode. Therefore, you should take care to perform these conversions as efficiently as possible.

On the positive side, writing the code to work with Unicode makes translation to multiple languages much easier. Other considerations to keep in mind are the availability of particular DLLs that may affect which libraries you can use and how you must link to them. If your application footprint becomes too large, you may have to resort to lower-level libraries. For instance, size can be reduced by not using MFC, but user interface code can be slightly more complicated.

12.2 Visual Studio.NET 2003

Windows CE .NET 4.2 is the newest version of the Windows CE operating system. The .NET suffix on the name refers to the inclusion of the .NET Compact Framework runtime that is bundled with Windows CE .NET.

It is important to note that the .NET Compact Framework is not the same as Windows CE .NET. Windows CE .NET implies Microsoft including embedded platforms in their strategic .NET initiative. Therefore, when building applications, either native (unmanaged) or managed applications can be created. To fully embrace .NET, the sample application outlined in this section, is written using Visual Studio .NET 2003 to target the .NET Compact Framework.

12.2.1 Set Up the Environment

In order to create applications that target Pocket PC 2003 you have the choice of the following tools:

- eMbedded Visual C++ 4.0
- Visual Studio .NET 2003

The optimal choice is to use Visual Studio .NET 2003. This is because it is a full-featured IDE with support for development of an expansive number of targets. From the same IDE .NET Compact Framework, .NET Framework, ASP .NET, SQL Server, and native applications (for desktop Windows only) can all be developed. If you are already familiar with the Visual Studio .NET IDE, then development for the .NET Compact framework will be almost identical. The only limitation will be the amount of functionality available via the .NET classes. Another advantage is the choice of languages you will have available. C# is the only language available that fully addresses .NET Compact Framework development as it was designed to take full advantage of that environment. It also has

several other miscellaneous benefits, such as automatic documentation generation, that are not available yet in the other .NET languages.

eMbedded Visual C++ 4.0 can be used to target the Windows CE .NET. However, it does not take advantage of the .NET Compact Framework. Therefore the applications that you create with it will be unmanaged. It mainly takes advantage of many of the new features available with Windows CE .NET, such as C++ exception handling.

For our second sample application, we will utilize the new .NET Compact Framework, and therefore use Visual Studio .NET 2003.

It is important to note that Visual Studio .NET 2003 can only be used to create managed applications, and eMbedded Visual C++ 4.0 can only be used to create native applications.

12.2.2 Creating the Project

Creating the stock quote project is fairly straightforward:

1. Launch Visual Studio .NET 2003
2. Click New Project
3. Select Visual C# Projects from the Project Types tree.
4. Select the Smart Device Application type on the right side of the dialog and click OK.

   A new dialog will be displayed allowing you to select the type of Smart Device Application you want to create.
5. Select Pocket PC and Windows Application and click OK. This will create a new WinForms project.

At this point your project should be open, displaying the form designer for the main window form. For VB developers, the form designer should look very familiar. For C++ developers the form designer should be a welcome sight.

The forms designer allows you to drag and drop controls onto the form window and then updates the properties of the controls from the Properties tab on the lower right corner of the screen. C++ developers are used to performing all these steps manually from within the code itself.

Under the covers, the appropriate code is automatically generated for you based on the properties, size, and window position of each of the controls

you manipulate. This code is generated in a function called ***InitializeComponent***. A call to this function is then automatically placed within the constructor for the class. This saves a significant amount of time in the initial development process. Instead of spending time writing code dealing with window sizes and positions, you can now concentrate on writing code to perform the application logic itself.

12.2.3 Porting the Code

Once you have added the appropriate controls to the form to make this application appear the same as the first sample, you will now be ready to start writing code to perform the stock-quote fetch.

The code that is created for the ***getquote_click*** handler is much simpler than the first sample application. You will first notice that you don't manually get the window text from the controls. It is simply a property that can be accessed. You will also notice there are no raw socket calls within the handler. The .NET Compact Framework makes a class called ***HttpWebRequest*** available. This is the class used to retrieve the stock quote. The program calls the ***Create*** function to create the HTTP request to the URL that is provided. Then the program calls the ***GetResponse*** function to retrieve the response from the request, which returns an ***HttpWebResponse*** object. The program can then use the properties of this object to get the price information for the stock ticker.

In the current version of the .NET Compact Framework there is an issue in the ***HttpWebResponse*** implementation that throws an exception due to an invalid HTTP header in the response from quote.yahoo.com. The sample application uses a workaround for this problem. Microsoft is aware of this problem and will be addressing it in a future release of the .NET Compact Framework.

12.2.4 Connection Management

Unfortunately, the Connection Manager API has not been exposed via a built-in .NET class. This means that in order to use it you will have to create a class that calls into unmanaged code using the interop services available with .NET. One of the interops, COM Interop, allows wrappers for COM interfaces to be automatically created and used. This is typically the easiest route to go if there is a COM interface exposed that you can use. However, the .NET Compact Framework does not currently support

COM Interop. Therefore, you can use the second choice, which allows us to call functions exported from any DLL.

In the case of our sample application, the functions required are exported from cellcore.dll. Therefore, go ahead and create the following definitions that allow you to call the API functions exported from cellcore.dll.

```
public class ConnMgrWrap
	{
		[DllImport("Coredll.dll")]
		public static extern UInt32 WaitForSingleObject (IntPtr handle,
UInt32 dwWait);
		[DllImport("Coredll.dll")]
		public static extern UInt32 CloseHandle (IntPtr handle);
		[DllImport("cellcore.dll")]
		public static extern IntPtr ConnMgrApiReadyEvent ();
		[DllImport("cellcore.dll")]
		public static extern Int32 ConnMgrReleaseConnection (IntPtr hConn,
UInt32 bCache);
		[DllImport("cellcore.dll")]
		public static extern Int32 ConnMgrConnectionStatus
			(IntPtr hConn, ref UInt32 dwStatus);
		[DllImport("cellcore.dll")]
		public static extern Int32 ConnMgrMapURL
			(string strUrl, ref System.Guid guidNet,
			ref UInt32 dwNextIndex);
		[DllImport("cellcore.dll",
EntryPoint="ConnMgrEstablishConnectionSync")] public static extern Int32
ConnMgrConnectSync
			(ref CONNMGR_CONNECTIONINFO connInfo, ref IntPtr ptrhConn,
UInt32 dwTimeout,
			ref UInt32 dwStatus);
				private static UInt32 WAIT_OBJECT_0 = 0;
		private static UInt32 CONNMGR_STATUS_CONNECTED = 16;
		private static UInt32 CONNMGR_PARAM_GUIDDESTNET = 1;
		private static UInt32 CONNMGR_PRIORITY_USERINTERACTIVE = 0x08000;
		private static UInt32 CONNMGR_FLAG_PROXY_HTTP = 1;
		private IntPtr m_hConnMgr = IntPtr.Zero;
		private bool m_bStarted = false;		~ConnMgrWrap ()

		{
			if (m_hConnMgr != IntPtr.Zero)
			{
				ReleaseConnection ();
			}
		}

		private static bool Success (Int32 hResult)
		{
			return hResult >= 0;
		}

		public bool Start ()
		{
			if (!m_bStarted)
			{
				IntPtr handleWait = ConnMgrApiReadyEvent ();
```

```
                        if (handleWait != IntPtr.Zero)
                        {
                                // Wait for our event for up to 60 seconds
                                UInt32 nWaitRet = WaitForSingleObject
(handleWait, 60 * 1000);
                                if (nWaitRet == WAIT_OBJECT_0)
                                {
                                        m_bStarted = true;
                                }
                                CloseHandle (handleWait);
                        }
                }
                return m_bStarted;
        }

        /// <summary>
        /// Uses the connection manager api to determine if we have
        /// a physical connection or not             /// </summary>
        /// <returns>true - there is a connection
        /// false - there is not a connection</returns>
        public bool IsConnected ()
        {
                bool bRet = false;
                if (Start ())
                {
                        if (m_hConnMgr != IntPtr.Zero)
                        {
                                UInt32 dwStatus = 0;
                                if (Success (ConnMgrConnectionStatus
(m_hConnMgr,
                                        ref dwStatus)))
                                {
                                        bRet = (dwStatus &
CONNMGR_STATUS_CONNECTED) > 0;
                                }
                        }
                }

                return bRet;
        }

        /// <summary>
        /// Attempts to create a physical connection using the connection
        /// manager apis
        /// </summary>
        /// <param name="strUrl">A url that describes the network we would
        /// like to connect to</param>
        /// <returns>true - connection was established
        /// false - connection could not be established</returns>
        public bool Connect (string strUrl)
        {
                bool bRet = false;
                if (!IsConnected ())
                {
                        CONNMGR_CONNECTIONINFO cmgrInfo = new
CONNMGR_CONNECTIONINFO ();
                        UInt32 dwIndex = 0;
                        if (Success (ConnMgrMapURL (
```

```
                                        strUrl, ref cmgrInfo.guidDestNet, ref
dwIndex)))
                                {
                                        // Now attempt to perform the sync.
Connection
                                        cmgrInfo.dwSize = (UInt32)Marshal.SizeOf
(cmgrInfo);
                                        cmgrInfo.dwParams =
CONNMGR_PARAM_GUIDDESTNET;
                                        cmgrInfo.dwPriority =
CONNMGR_PRIORITY_USERINTERACTIVE;
                                        cmgrInfo.dwFlags = CONNMGR_FLAG_PROXY_HTTP;
                                        cmgrInfo.bExclusive = 0;
                                        cmgrInfo.bDisabled = 0;
                                        cmgrInfo.lParam = 0;
                                        cmgrInfo.hWnd = IntPtr.Zero;

                                        UInt32 dwStatus = 0;
                                        Int32 nRet = 0;
                                        UInt32 dwTime = 30*1000;
                                        if (Success (nRet = ConnMgrConnectSync (ref
cmgrInfo,
                                                ref m_hConnMgr, dwTime, ref
dwStatus)))
                                        {
                                                bRet = (dwStatus &
CONNMGR_STATUS_CONNECTED) > 0;
                                        }
                                }
                        }
                        return bRet;
                }

                /// <summary>
                /// Releases the connection. Does not necessarily disconnect, but
                /// will if no other applications have an open handle to the same
                /// physical connection
                /// </summary>
                /// <returns>true - successfully released the connection
                /// false - could not release the connection</returns>
                public void ReleaseConnection ()
                {
                        if (m_hConnMgr != IntPtr.Zero)
                        {
                                ConnMgrReleaseConnection (m_hConnMgr, 1);
                                m_hConnMgr = IntPtr.Zero;
                        }
                }
        }

        [StructLayout (LayoutKind.Sequential)]      public struct
CONNMGR_CONNECTIONINFO
        {
                public UInt32 dwSize;      // native DWORD
                public UInt32 dwParams;    // native DWORD
                public UInt32 dwFlags;     // native DWORD
                public UInt32 dwPriority;  // native DWORD
                public UInt32 bExclusive;  // native BOOL
                public UInt32 bDisabled;   // native BOOL
```

```
            public System.Guid guidDestNet; // native GUID
            public IntPtr hWnd; // native HWND
            public UInt32 uMsg;        // native UINT
            public UInt32 lParam;      // native LPARAM
            public UInt32 ulMaxCost;   // native ULONG
            public UInt32 ulMinRcvBw;  // native ULONG
            public UInt32 ulMaxConnLatency; // native ULONG
      }
```

The first part of the class above describes several functions that are exported from unmanaged DLLs.

- The ***DllImport*** attribute is used to tell the C# compiler that the following function definition must be imported from the specific DLL.
- The ***WaitForSingleObject*** and ***CloseHandle*** calls are required to make sure the Connection Manager is ready and to close the handle once the ready event has been fired.

The remainder of the functions should be familiar; they are the same functions used in the C++ sample application. The remainder of the class just describes the C# functions that will be called from the ***StockTicker*** class when attempting to make sure there is a physical connection ready before any attempts to retrieve a stock quote. Now you can create a new instance of the ***ConnMgrWrap*** class inside the ***StockTicker*** class.

Before attempting to retrieve a stock quote, ***IsConnected*** is called. If ***IsConnected*** returns false, then ***Connect*** is called to attempt to make the physical connection. You will also notice in the destructor of the ***StockTicker*** class, ***ReleaseConnection*** is called. This ensures that when the application is shut down, the Connection Manager closes the connection if no other applications are currently using it.

12.2.5 Summary

As you can see from this sample application, it contains much less code and was easier to create than the first one. Much of this is due to the form designer and the overabundance of well-designed classes that the compact framework makes available. For instance, much of the windowing information is easily available via class properties.

There are also many classes such as ***HttpWebRequest*** that can simplify common tasks. Deployment can also be simplified, since there is no processor-specific code generated. This means that a single binary can be deployed on all supported devices. However, if native code is used, such as the Connection Manager, then that native code must be ported to all

devices. Fortunately, the Connection Manager (cellcore.dll) is part of the operating system, and so this is not an issue for the sample application. But if your application needed to use an in-house DLL that DLL would have to be made available for all devices you intend to support. Consequently, this limits the benefit of deployment simplicity.

As the sample illustrated, there are also many pitfalls. If functionality has not been put directly into the .NET Compact Framework, like the Connection Manger, then you must wrap this functionality into the application. This can be error prone and difficult to debug. Many VB developers are probably familiar with this issue.

There is also the issue of performance to consider when using the .NET Compact Framework. Since the IL code must be compiled at runtime to native code, this makes load-up times slower, and in many cases, the native code that is generated will also be significantly slower. Microsoft does claim, however, that for most code, the speed will be nearly identical to what a native compiled binary would be.

In conclusion, developing for the .NET compact framework can save considerable time and just like any new technology, there is a learning curve that must be taken into consideration before gaining confidence in being able to make good development decisions.

12.3 ASP.NET

ASP.NET is not a technology specific to mobile application development. It allows developers to create Web content and services that utilize the .NET framework. However, much of the design of ASP.NET is centered on mobile application development.

12.3.1 Choosing ASP.NET

A developer would typically choose to develop an ASP.NET Web-based application for several reasons. Intranet applications lend themselves well to ASP.NET. ASP.NET also allows the system administrator to simply deploy the application on a single Web server. This eliminates costly installations to perhaps hundreds of computers and devices. If the application utilizes a database for its functionality, an ASP.NET Web application can greatly simplify application development. Instead of each client device connecting directly to a database, they can connect to the Web server. The Web server can then manage the communication with

the database. This is especially beneficial when the devices are using connections such as GPRS/EDGE to access the application functionality.

The stock-quote application itself could be Web-based. The user could then just connect to page and retrieve quotes. This would also eliminate portability and deployment issues.

It is important to note that Pocket Internet Explorer does not support all HTML capabilities. The following table lists supported and unsupported features.

Table 5: Supported and Unsupported Browser Features

Supported Features	HTML 4.01 Extended HTML (XHTML), XML Cascading Style Sheets (CSS) WAP support, WSP, WML 2.0, WMLSCRIPT, WBMP ActiveX support, with sourced events Enhanced scripting and Document Object Model (DOM) support New extensible imaging library No auto-download, must be preinstalled Not affected by "Fit to Screen" option Macromedia Flash Image Maps Framesets Security, SSL 2.0 and 3.0 40 Bit encryption 128 Bit encryption (add on for PPC 2000) Microsoft Jscript version 5.5
Unsupported Features	DHTML full implementation Client-side VBScript Animated GIFs Multiple Windows Java applets

12.3.2 Advantages and Disadvantages

The following advantages are gained by using ASP.NET:

- **Eliminates Deployment Issues:** Your application does not have to be deployed to every machine that is going to use it. It must simply be copied to a Web server or a Web farm for deployment.
- **Portability:** The application does not have to be compiled for each mobile processor with which you intend to deploy. Since only HTML and script are returned to the client, any device with a browser can access and use the application.

- **Speed of Development:** Since the application does not have to be tested and debugged on multiple devices, development time is reduced. And because you are relying on the browser as the user interface engine, there is no client code to create.

There are, however a number of disadvantages:

- **Constrained User Interface:** Since you do not have all the user interface elements with HTML that you have on Windows CE itself, the user interface cannot be as elaborate. This can typically result in a user interface that is slow and cumbersome, since many steps may be required to complete the same task.
- **Performance:** Since the entire user interface is rendered in HTML over a potentially slow network connection, performance will be significantly slower.
- **Requires Network Connection:** In order for your application to be useful, the user must be able to establish a connection to the Web server. If you intend for your application to also be usable in an offline state, then a Web-based approach is not suitable.

12.3.3 Application Porting

Porting your application to ASP.NET can be simple or it can be complex. In the case of the stock quote sample, it would be fairly straightforward, using the following steps:

1. Create the Web form layout
2. Add the same controls you had in the CE application
3. Copy in code that retrieves quote data from the stock quote provider
4. Create appropriate error pages

Fortunately, ASP.NET will handle generating the markup. This means that based on the request, it can generate the appropriate HTML or WML for the device, and size it appropriately.

However, if your application is more complicated, you will probably have to redesign the flow of the user interface before you attempt to port it. Once you have redesigned the user interface elements, you can then begin to port the application. Fortunately ASP.NET does provide many controls specifically for mobile development. These are called the ASP.NET Mobile Controls, formerly known as the Microsoft Mobile Internet Toolkit

(MMIT). These controls extend the power of the .NET framework and Visual Studio .NET to build mobile-Web applications by enabling ASP.NET to deliver markup to a wide variety of mobile devices. For more information on the usage and power of ASP.NET mobile controls visit http://www.asp.net/mobile/intro.aspx.