ONE

Introduction

Around twenty years ago a computer revolution started when the IBM PC was released. The IBM PC took computing away from the air-conditioned environment of the mainframe and minicomputer and put it onto the desk of potentially everyone. Nowadays most workers have a PC on their desk, and many have a PC at home, too. Laptop computers allow users to have one computer that can be used both at home and at work, as well as on the road. PCs are generic computing devices providing tremendous computing power and flexibility, and all PCs in the world from laptops to desktop PCs and through to servers have fundamentally the same architecture. Living through the PC era has been fun, frustrating, and exciting. However, there is an even bigger revolution on the way with more potential and even more challenges—the move to truly mobile-device-based computing.

In the last few years computing devices have been coming onto the market that provide unparalleled portability and accessibility. Microsoft Windows CE devices, such as the palm-size device and handheld PC, provide cutdown desktop PC capabilities in a really small footprint, and Palm Pilot has been a very successful PDA (Personal Digital Assistant). Microsoft Pocket PC has tremendous features for enterprise computing, games, and entertainment. The Windows CE operating system has been embedded into many appliances (such as gas pumps and productions systems) for monitoring and control. Unlike the generic PC, these computing devices are not all the same and are designed for specific purposes.

We think of laptop PCs as being mobile devices, but really they are a convenient way of moving a PC from desktop to desktop. Think of a situation where I go to a client's offices, and as I walk through the door I want to check the names of the people I will be meeting. With a laptop computer, I have to power-on (assuming I haven't let the battery run down), wait for the operating

system to boot, login, run my calendar application, and look up the information. This whole operation could take five minutes during which I have to suffer quizzical looks from the receptionist. The same scenario with a true mobile device is entirely different—with instant power-on and one-click access to my calendar, I can have the information within 30 seconds.

Most people tend to think of a mobile worker as the typical road warrior, out of the office taking orders from customers and flying or driving from here to there and never visiting the office from one week to the next. Sales force automation (SFA) and field engineer support are classic applications for this type of activity. The reality, though, is that we are *all* mobile workers—start thinking of a mobile worker as someone away from his or her desk. If I am at a project status meeting, I may be expected to take decisions or provide comments on a project's progress. I need to have the information in front of me, but chances are it is on my desktop PC back in the office. With a mobile device, I can bring the information with me.

The mobile devices are designed to fill in the gaps in our lives where we haven't had convenient access to computing. The desktop PC provides computing capability at the desk at work and at home. Mobile devices allow access to computing while commuting and traveling, at client meetings, on holidays, and anywhere else we may be. Computing is not just about work, so these devices can also entertain. I can listen to my favorite music, play a game, or read a book.

To date, most devices have worked their way into organizations through personal purchases. The devices arrive in the office on Monday morning and are hooked up to the desktop PC; information such as contacts and tasks are then downloaded onto the device. Of course, this doesn't always work the first time, so IT support staff are called in to try to support a device that may be new to them. Consequently, many organizations are now starting to produce strategies for adopting and supporting mobile devices. It soon becomes apparent that these devices should be enterprise players and have the capability of downloading, uploading, and manipulating data from databases, the Internet, and the intranet.

Mobile devices are not just about mobility. For example, desktop Windows CE devices are available that provide thin-client computing. They have Windows Terminal Server client installed, allowing them to effectively run Windows NT and 2000 applications. Being thin clients, they are easy to set up, configure, and maintain. Windows CE has successfully been embedded into many different custom devices by developers around the world.

As devices are produced which combine technologies, the possibilities become even more exciting. Combining a computing device with a GSM phone allows mobile computing with access to data even when a telephone connection is not present. Enterprise servers can push data down onto the devices without user intervention—the device will even wake itself up to receive the data. By incorporating GPS (Global Positioning System) support, a device's location may be determined very accurately, and this can be used to direct the

user to a local service, such as a coffee shop or gas station. Harnessing these possibilities requires applications, and this book shows how to do just that using the Windows CE operating system.

About Microsoft Windows CE

First, let's start by describing some of the Windows CE operating system characteristics and capabilities:

- Compatible API with Windows NT and 2000
- Multiprocessing, multithreaded support with synchronization
- Virtual memory architecture
- File system and property database support
- TCP/IP stack with functions allowing HTTP and socket communication
- Access to Windows NT and 2000 network resources
- Serial port communications
- Database access through ADOCE (ActiveX Data Objects for Windows CE)
- COM (Component Object Model) support for building componentized software
- DCOM (Distributed Component Object Model) support for building Windows DNA client software
- Synchronization of data with desktop PCs using ActiveSync

Windows CE is a modular operating system designed to build computing devices. Its modularity means that engineers can select which parts of the operating system are required—for example, a device may not need a keyboard or a display, but perhaps it needs networking capability. By selecting only those modules a device requires, the size and cost of the device can be controlled. Device manufacturers can use the Microsoft Platform Builder product to produce their own customized devices, or use one of the standard configurations such as the Pocket PC or Handheld PC. These standard configurations come with utilities and tools, such as Pocket Word or Pocket Internet Explorer, that can be incorporated into the devices.

This flexibility also produces problems for the application developer. While the Windows CE operating system may support some functionality, such as a TCP/IP stack, the device being targeted may not. Therefore, the application developer should first determine if the feature is present before programming for it!

There is currently much confusion around Windows CE versions and naming conventions. In particular, recent devices such as the Pocket PC are labeled "Powered by Windows" and don't actually mention Windows CE at all. The truth is that Pocket PC does use Windows CE. Here are some of the more recent releases of Windows CE:

- Windows CE 3.0. This version of the operating system is designed to provide hard, real time operating system characteristics and other improvements. Pocket PC uses this version of Windows CE.
- Windows CE 2.12. Used primarily by embedded device manufacturers using the Microsoft Platform Builder product. This version did not make its way into many consumer devices.
- Windows CE 2.21. The version of Windows CE used in Windows Handheld and Palm size devices.

To add to the confusion, each of the standard configurations such as palm-size and handheld devices has its own version number. For example, the Handheld PC Edition Version 3.01 actually runs on Windows CE 2.11. To simplify matters, the descriptions of devices will apply to the following operating systems:

- Pocket PC—Running on Windows CE 3.0
- Handheld PC—Running on Windows CE 2.11
- Palm size PC—Running on Windows CE 2.11

Microsoft Pocket PC

The Pocket PC does not have a keyboard and supports written character input using SIP (Supplementary Input Panel) with either character recognition or a virtual keyboard. Pocket PC can also use Microsoft Transcriber, a program that uses neural network programming techniques for handwriting recognition. Pocket PC provides multimedia playback (for music using MP3 and video), Microsoft Reader for reading books, Microsoft Pocket Word and Pocket Excel, and Microsoft Pocket Internet Explorer for web access.

Pocket PC marks the start of a new era in mobile devices. Not only does it offer unparalleled consumer functionality; it also provides tools for the enterprise developer for accessing databases, the Internet and intranet, and server-side components.

Most Pocket PC devices support either a type-1 or type-2 Compact Flash slot which can be used for expanding storage (using either solid-state memory devices or Winchester disk drives), or adding peripheral support such as barcode readers, cameras, modems, or connections to GSM mobile phones.

Handheld PC

The Handheld PC differs from Pocket PC primarily in its keyboard support. It also has a larger screen. Sub-notebook size devices with larger screens and keyboards are also available.

Handheld devices often support a full-size PCMCIA card and a Compact Flash card slot and may have an inbuilt modem. This device configuration is best suited to job functions that require large amounts of data entry and bet-

ter display capabilities, such as customer-facing situations. Either the screens are touch sensitive, or some form of mouse support is provided. The sub-notebooks running Windows CE are generally the same size as some of the smaller Windows 98 laptop computers, and there is less cost differential.

Palm Size PC

The Palm size PC has been largely replaced with the Pocket PC. It provides a user interface that is more similar to Windows, as opposed to Pocket PC, which is more like a browser interface. The Palm size PC suffered from poor battery life and insufficient capability.

About This Book

First, let me state what this book is *not!* This book does not look at user interface programming. Why not? I wanted to concentrate on the behind-the-scenes operating system facilities that are used to make really great applications. There are many good books on programming the user interface, and many of the principles and techniques are the same on Windows CE as for Windows NT/98/2000. The major difference is the smaller size of display, and knowing which user interface features are supported.

While many of these operating system features are similar to counterparts on Windows NT/98/2000 and often use the same API (Application Programming Interface) functions, the emphasis is different. Windows CE applications need to communicate. They need to communicate with other devices, to communicate with the Internet, to communicate with databases, and to communicate with server-side components. These are the areas on which I concentrate.

Also, these devices are smaller and have less memory in which to execute applications and to store data. Writing memory-efficient applications that can degrade gracefully in low-memory situations is essential. Data storage can be in files or in databases, and Windows CE provides unique techniques for both. These issues are covered in this book.

The techniques here can be used in nearly all Windows CE devices, including standard devices such as Pocket PC and Handheld PC, and customized embedded Windows CE devices produced by embedded developers. Probably 90 percent of the techniques here work in Windows CE 2.11 or 2.12. I have pointed out code that is specific to Windows CE 3.0 and Pocket PC in particular.

I have tried to provide plenty of code samples showing how to use the features being discussed. There is little or no user-interface code to get in the way of seeing the really important code. Feel free to take the code (it is on the CDROM) and incorporate it into your own applications. However, *please, please, please* add error-checking code. For the sake of brevity it is omitted from the source code samples, but it is essential in any production code.

About You

I expect that you are a developer about to start a serious Windows CE application development project for Pocket PC, or an embedded Windows CE developer who needs to write applications to run on a custom device, or perhaps someone who wants to find out more about the innards of Windows CE, or perhaps just plain inquisitive—it really doesn't matter. However, to get the most out of the book you will probably need the following experience:

- C and C++ knowledge. Most of the code samples are written using C; a few require C++ specific knowledge.
- Some Windows API programming experience. You should have already written some Windows applications, perhaps on Windows NT, 98, or 2000.
- Experience using a Windows CE device. You should try using a Windows CE device for a while before attempting to write or design applications for a device. You will need to become accustomed to the capabilities, limitations, and different way of doing things.

I hope that after reading this book you will know a lot more about Windows CE programming in particular, and more about programming in general.

About MFC (Microsoft Foundation Classes) and ATL (ActiveX Template Libraries)

This book is primarily about using the Windows CE API functions, so most of the code is standard C code calling these functions. If you are writing an application using MFC, you will be able to call these functions in exactly the same way. However, there are times when MFC provides classes that make calling these functions easier and more efficient. For example, the Windows CE property databases can be programmed through direct API calls, but the MFC classes make writing database applications much easier. This book will show how to use MFC classes when appropriate.

Many developers are now writing components using ATL. This can be a difficult learning process, but the benefits are great. ATL is mainly based around writing and using COM components, although ATL can also be used to write applications. This book does not use ATL to any great extent, but as with MFC, the API calls and techniques can be incorporated into ATL applications and components.

eMbedded Visual C++ 3.0

In the past, Microsoft has provided add-ins for Visual C++ to provide a Windows CE development environment. The main problem with the add-ins was

that all the facilities used for developing Windows NT/98/2000 applications were still present. Also, tools like the dialog editor were not tailored to writing Windows CE applications. The documentation was difficult to follow—Windows CE-specific comments were embedded in the full MSDN documentation set.

eMbedded Visual C++ 3.0 (Figure 1.1) is a new tool specifically designed to write Windows CE applications. It is based on Visual C++ and shares the same user interface, but only those tools and facilities necessary for writing Windows CE applications are present. The 'WCE Configuration' toolbar provides drop-down combo boxes that allow selection of the target platform (for example, Pocket PC, Palm size PC 2.11, or H/PC Pro 2.11); the target CPU (such as ARM and MIPS); whether the build is debug or release; and the type of device to be run on (for example, emulation or a target device).

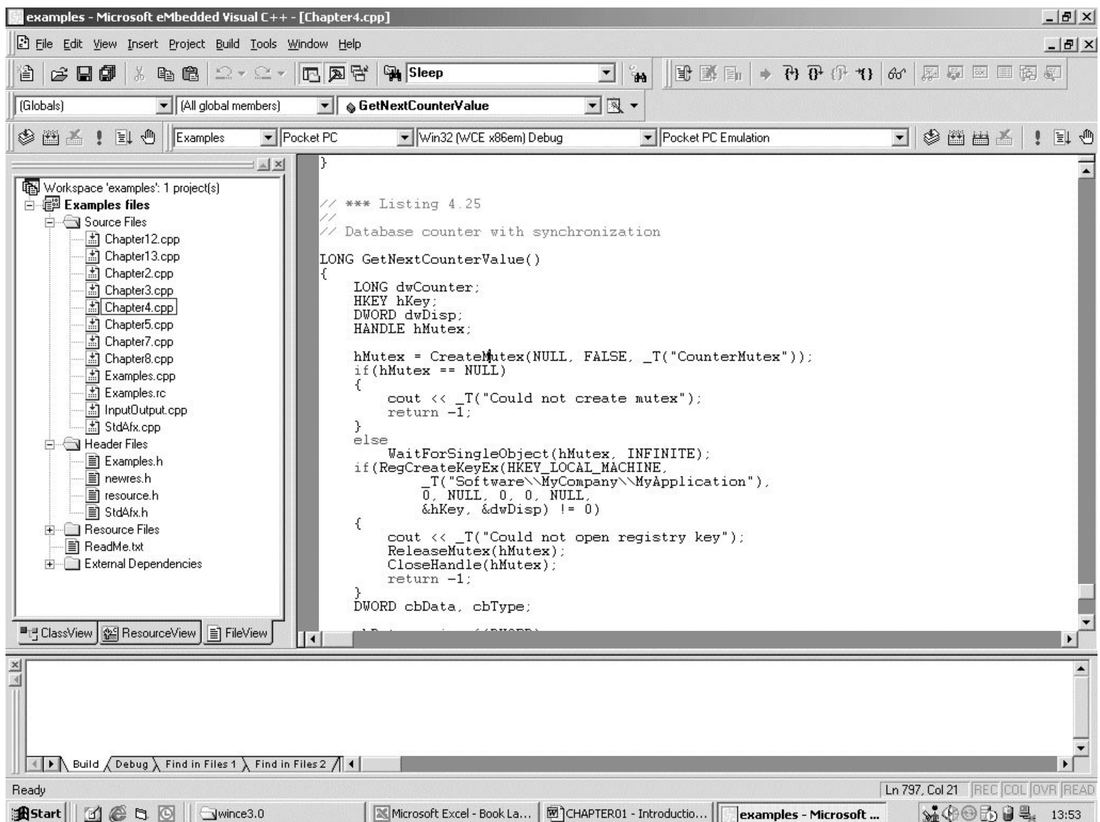

Figure 1.1 *eMbedded Visual C++*

API Reference

Sleep

This function suspends the execution of the current thread for a specified interval.

```
void Sleep(
DWORD dwMilliseconds );
```

Parameters

dwMilliseconds
Specifies the time, in milliseconds, for which to suspend execution. A value of zero causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution. A value of INFINITE causes an infinite delay.

Return Values

None.

Remarks

A thread can relinquish the remainder of its time slice by calling this function with a sleep time of zero milliseconds.

You have to be careful when using **Sleep** and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses **Sleep** with infinite delay, the system will deadlock. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx**, rather than **Sleep**.

Requirements

Runs on	Versions	Defined in	Include	Link to
Windows CE OS	1.0 and later	Winbase.h		
Pocket PC	Windows CE OS 3.0	Winbase.h	Windows.h	

Figure 1.2 *Typical help screen for a Windows CE function*

The documentation is specific to writing for Windows CE and details carefully how the various functions are implemented in the various operating system and platform versions. Figure 1.2 shows a typical example for the Sleep function.

eMbedded Visual C++ allows you to write Windows CE application for any target device for which you have an SDK (Software Development Kit). As well as producing a customized Windows CE operating system, the Microsoft Platform Builder can also produce an SDK for that device. The SDK can then be installed in eMbedded Visual C++ and applications can be developed for the device.

All the sample projects covered in this book and distributed on the CD-ROM are eMbedded Visual C++ projects, and should not be compiled using the standard Visual C++. Workspaces in eMbedded Visual C++ use the `.vcw` extension, and projects the `.vcp` extension. In Visual C++ `.dsw` and `.dsp` are used. eMbedded Visual C++ can import Windows CE projects created using Visual C++. However, in my experience it is sometimes better to rebuild the project and import the files.

Common Executable Format (CEF)

One of the downsides to writing Windows CE applications in the past was the number of different microprocessors that needed to be supported, such as MIPS and SH3. Starting with Pocket PC, Windows CE devices now support a processor-neutral machine code set called Common Executable Format, or CEF (pronounced 'keff'). You can compile into CEF using eMbedded Visual C++ and then run that single executable on any platform that supports CEF, such as Pocket PC.

CEF-enabled platforms have a translator that takes the CEF code and translates it into the native code, such as MIPS or SH3. Translation can take place every time the application is run, or the converted code can be saved. There is an overhead in performance—CEF applications run at around 80 percent the speed of native applications.

Emulation Environments

Many Windows CE SDKs, such as Pocket PC, support an emulation environment that runs on the desktop PC. This can be used to test and debug your applications and is generally quicker to use than downloading applications onto a real device. However, you should not solely rely on emulation for testing for the following reasons:

- Emulation is not perfect, and applications that run under emulation may not work correctly on a proper device. Facilities such as networking and RAS dialup connections may behave differently.
- User interfaces may appear differently under emulation, since there are differences in how standard controls and fonts are implemented.
- Desktop performance is generally much better than on a real device. Applications may perform adequately under emulation, but run too slowly on a Windows CE device.

Using emulation does save large amounts of development time, particularly when you are debugging non-user-interface code.

The Code Samples

Throughout the book you will find code samples showing how to use the facilities being discussed. All the code is on the CDROM, so it can be copied directly into your application. Unless otherwise stated, all the code is in a single project called `examples.vcp` in the directory `\examples`. The source code is arranged by chapter, and each chapter has its own source file, for example `Chapter2.cpp`, `Chapter3.cpp`, and so on.

The `examples.exe` application can be run on a real device, or under emulation. The user interface has been optimized to run under Pocket PC, but

Figure 1.3 *Examples application used to run sample code*

can easily be adapted to run on other platforms. The menu contains drop downs for each of the chapters arranged into groups, and the drop downs contain menu items allowing each code sample to be run. Figure 1.3 shows how the application looks, with sample output. No prizes for best user interface here! Note that not all the sample code will run on all platforms. In particular, some samples will not run under emulation.

The code samples are designed to remove all irrelevant code so you can concentrate on what is really important. In the Examples project, all output is displayed to a read-only edit window (which, in Figure 1.3, contains the text "Mounted vol: SystemHeap" and so on). A C++ class object called '`cout`' has been created to emulate the basic behaviors of the standard C++ '`cout`' object used in command line, character mode applications. The '`cout`' object is an instance of the class `COutput` which is declared in `Examples.h` and implemented in `InputOutput.cpp`. The '<<' operator has overloads for most common data types, including strings, integers, and characters. Calling the `COutput` 'CLS' method removes all the text from the text edit window. You will find statements like the following to display data to the edit window:

```
cout << _T("Unicode File") << endl;
```

Input is obtained from the user in a dialog using the function `GetTextResponse`. The function is passed the string to prompt the user with, a string in which the data will be returned as well as the maximum number of characters of data that can be placed in the string. The function returns `TRUE` if a string is returned, or `FALSE` if the user pressed Cancel.

```
if(!GetTextResponse(_T("Enter URL to Display: "),
              szURL, MAX_PATH))
  return;
```

The function `GetFileName` will display a File Open command dialog box allowing the user to select a file. This function takes the same arguments as `GetTextResponse`:

```
if(!GetFilename(_T("Enter filename:"),
              szFilename, MAX_PATH))
  return;
```

Some of the sample code is in separate projects, and because some of these projects run on a desktop PC, the projects should be compiled using Visual C++ 6.0.

Unicode Text and Strings

Before starting out there are a couple of topics that need to be covered, and the first of these is Unicode. Windows 98 API functions have partial support for Unicode strings, and Windows 2000 and NT allow applications to call either Unicode or ANSI versions of the API functions. Windows CE, on the other hand, only supports Unicode, so you will need to write your applications using Unicode strings and text.

Most of us grew up safe in the knowledge that a character was stored in a single byte using eight bits. Character strings are stored in '`char`' arrays and are terminated with a `NULL`, ANSI 0 character. Strangely enough, the '`char`' data type is signed, but we get used to that. The problem is that there are many more than the 255 characters that fit in a '`char`' used by different languages around the world, so tricks need to be employed to support all these characters. Two such tricks are:

- Use multi-byte character strings (MBCS), where special characters act as lead-ins indicating that the next character should be treated as an entirely different character.
- Use Code Pages, in which the same ANSI character number is used to display completely different characters depending on which code page is loaded.

Neither of these tricks is satisfactory. Parsing MBCS strings is difficult; for example, the length of a string can only be determined by traversing the entire string and inspecting each character. With code pages, you can display completely incorrect text by having the wrong code page loaded for the text being displayed. The Unicode solution uses two bytes to store a single character. This allows up to 65536 different characters to be displayed—more than enough for all the languages around the world. With Unicode, a character is stored as an

unsigned two-byte integer value. They are also known as 'wide byte characters'. The Unicode characters in the range 0x00 to 0x7F are reserved for ANSI characters, so ANSI characters always have the high byte set to zero when represented in Unicode.

Compilers do not provide native support for Unicode—that is, there is no magic compiler switch that changes a `char` from one byte to two bytes. Instead, support for Unicode is achieved through defines and typedef statements in header files. The data type `wchar_t` is used to represent a Unicode character, and an array of `wchar_t` is used to store strings. As with ANSI strings, a NULL terminates a string, but this is a two-byte rather than a one-byte value. ANSI strings and characters can be used alongside Unicode strings and characters—you can continue to use the '`char`' data type. This is important because data coming from the outside world (through the Internet or as a file) may use ANSI characters, and these need to be converted before being used.

Unicode characters obviously take twice as much space as ANSI to store strings. In many applications the majority of strings stored using Unicode actually store ANSI characters, so every other byte is a `NULL`. In Windows CE, the compression algorithms used to store data in the object store (that is, data stored in files or databases) are optimized to recognize this sequence.

Generic String and Character Data Types

You can use the standard Unicode data type `wchar_t`, but it is more usual to use generic string data types, and then use compiler defines to specify which character type should be used for the compilation. You can write code that can be compiled for ANSI and Unicode and is portable. The define `_UNICODE` is defined either as a compiler switch or using `#define` to indicate that the Unicode version of API functions should be used. Some header files expect the `UNICODE` define to be used, so both often end up being defined. The compiler defines `_MBCS`, and multi-byte character strings (MBCS) are used in Windows NT/98/2000 to compile for ANSI characters but are not supported under Windows CE. If neither `_MBCS` nor `_UNICODE` is defined, the header files default to single-byte character strings (SBCS). SCBS don't use lead-in characters to extend the supported range of characters.

To use generic string and character data types, include the file `tchar.h` and ensure that `_UNICODE` or `_MBCS` is defined as appropriate. To declare a character, use the data type `TCHAR`, and this will be compiled to `wchar_t` or `char` depending on the define in operation. The following code declares a character variable and a character string that can store up to ten characters including the terminating `NULL`:

```
TCHAR cChar;
TCHAR szArray[10];
```

Rather than using the `LPSTR` data type for specifying a pointer to a character string, you should use `LPTSTR`. This will be compiled to either a '`char*`' or a '`wchar_t*`'.

String Constants

In the following code fragment, the string constant "my string" will always be compiled as an ANSI character string constant using one byte per character.

```
LPTSTR lpszStr = "my string";
```

You will get a compiler type mismatch error if you try to compile this code with _UNICODE. The header file tchar.h declares two macros '_T' and '_TEXT' that are used to specify Unicode character string constants when _UNICODE is declared, and ANSI character string constants when _MBCS is declared. So, the previous line of code should be written as

```
LPTSTR lpszStr = _T("my string");
```

or

```
LPTSTR lpszStr = _TEXT("my string");
```

The L macro can be used to force a Unicode string constant. In this next line of code, the LPWSTR data type declares a Unicode string pointer and points it to a Unicode string constant.

```
LPWSTR lpszStr = L("my string");
```

With Windows CE programming you will need to use the _T or _TEXT macro around just about every string constant. My preference is for _T, only because it is shorter. I like to set up an eMbedded Visual C++ macro and assign it to the Ctrl+T key sequence to generate the _T("") sequence in the source file. To do this:

- Select the **Tools+Macro** menu command.
- Enter the name of the macro, say 'T', and click the **Record** button.
- Enter the text _T("") into a source file, followed by two left arrow key presses to locate the cursor between the two double quotes.
- Turn off recording by pressing the Macro toolbar icon with a square box.
- Select the **Tools+Macro** menu command again, this time to assign the macro to a keystroke.
- Select your macro from the list and click the **Options** button.
- Select the **Keystrokes** button, and assign the macro to the required keystroke, for example Ctrl+T.

Macros in eMbedded Visual C++ are recorded using VB Script. Here is the source for the _T macro:

```
Sub T()
'DESCRIPTION: A macro to enter _T("") into a source file.
'Begin Recording
  ActiveDocument.Selection = "_T("""")"
  ActiveDocument.Selection.CharLeft dsMove, 2
'End Recording
End Sub
```

Calculating String Buffer Lengths

One of the most common bugs introduced when moving to Unicode programming concerns calculating buffer lengths—all too often, code assumes that characters are stored in one byte. For example:

```
TCHAR szBuffer[200];
DWORD dwLen;
dwLen = sizeof(szBuffer);
```

We might expect `dwLen` to contain the value 200, but it will actually contain 400, which is the number of *bytes* occupied by `szBuffer`. If `dwLen` were passed to a function indicating how many *characters* can be placed in `szBuffer`, the application might fail, as the function could exceed the bounds of the array `szBuffer`. The following code should be used instead, and this will work for both ANSI and Unicode compilation.

```
dwLen = sizeof(szBuffer) / sizeof(TCHAR);
```

When passing the length of a string buffer to a function, check whether the function expects the size of the buffer in bytes or characters.

Standard String Library Functions

We are all accustomed to the standard C run-time functions for string manipulation—`strlen`, `strcpy`, and so on. These functions work with the '`char`' data type and cannot be used for Unicode strings. Unicode equivalent functions are provided, such as `wcslen` and `wcscpy` (standing for 'wide character string length,' and 'wide character string copy').

Generic string functions are also available which will be compiled to the ANSI or Unicode function equivalents. For example, the function `_tcslen` will compile to `strlen` if `_MBCS` is defined, or `wcslen` if `_UNICODE` is defined. The header file `tchar.h` should be included to enable this behavior. Using the `_tc` functions makes code portable between ANSI and Unicode. The samples in this book tend to use the `wcs` functions rather than `_tc`, since I never intend to port this code away from Unicode. Table 1.1 shows some of the C common run-time string functions and their generic and Unicode equivalents.

Converting Between ANSI and Unicode Strings

There are times when you will need to convert ANSI strings or characters to Unicode and vice versa. Examples include:

- Reading an ANSI text file into a Windows CE application
- Reading and writing characters from a serial device that supplies data in ANSI
- Reading and writing data from Internet servers, such as web or email servers, most of which expect text in ANSI

Table 1.1 *C common run-time string functions with generic and Unicode equivalents*

Purpose	Generic String Function	ANSI Function	Unicode Function
Return length of string in characters	`_tcslen`	`strlen`	`wcslen`
Concatenate strings	`_tcscat`	`strcat`	`wcscat`
Search for character in string	`_tcschr`	`strchr`	`wcschr`
Compare two strings	`_tcscmp`	`strcmp`	`wcscmp`
Copy a string	`_tcscpy`	`strcpy`	`wcscpy`
Find one string in another	`_tcsstr`	`strstr`	`wcsstr`
Reverse a string	`_tcsrev`	`_strrev`	`_wcsrev`

Converting an ANSI character to Unicode is easy—all you need to do is set the high byte in the Unicode character to zero and copy the ANSI character into the low byte. In this next code fragment, the `MAKEWORD` macro combines a low byte and high byte into a single two-byte word, and the result is assigned to a Unicode character.

```
WCHAR wC;
char c = 'C';

wC = MAKEWORD(c, 0);
```

You can convert string using one of the C run-time functions:

- `mbstowcs`—Convert a multi-byte (ANSI) string to wide character string (Unicode)
- `wcstombs`—Convert a wide character string to multi-byte string

Both of these functions take three arguments that are the buffer in which to place the converted string, the string to convert, and the maximum number of characters that can be placed in the string. Both functions return the number of converted characters placed in the string. The following code converts an ANSI string to Unicode and a Unicode string to ANSI.

```
WCHAR szwcBuffer[100];
char szBuffer[100];

char* lpszConvert = "ANSI String to convert";
WCHAR* lpszwcConvert = _T("Unicode string to convert");
int nChars;

nChars = mbstowcs(szwcBuffer, lpszConvert, 100);
nChars = wcstombs(szBuffer, lpszwcConvert, 100);
```

If you are using code pages, the Windows API functions `MultiByteToWideChar` and `WideCharToMultiByte` should be used since you can specify the target or destination code page to be used for the conversion.

Error Checking

As with any operating system, it is imperative to check the return results when calling Windows CE API functions—never assume that the function works. Many of the code samples in this book do not have sufficient error-checking code for use as production code, so you will need to add it if you take code from this book for use in your own applications.

Nearly all Windows CE API functions return a value indicating success or failure, but little information detailing the nature of the error. You should call the function `GetLastError` to determine the actual error number encountered. You can look up the error numbers in the header file `winerror.h`, where you will find a short description of the error. This file is located in the "`\Windows CE Tools\wce300\MS Pocket PC\include`", or another folder appropriate to the SDK version you are using. The on-line documentation often lists the common errors encountered when calling specific Windows CE functions.

Windows CE devices, unlike Windows NT/98/2000, do not support the `FormatMessage` function for producing textual descriptions of error numbers, but the function does work under emulation—watch out for this one.

Adding comprehensive error-checking code can increase significantly the size of your application's code. With memory-tight Windows CE devices, this can be a problem. You should therefore place debug-specific error-checking code in `#ifdef / #endif` compiler directives with the `_DEBUG` define so that the code will not be included in your released application.

```
#ifdef _DEBUG
  // perform error checking that does not need to be in
  // the production version
#endif
```

Exception Handling and Page Faults

A page fault occurs when an application attempts to read or write data from or to a page that does not have memory associated with it, or to a memory address that is illegal. If you try to execute the following code on a desktop PC, you will get an unhandled page fault error box, and your application will terminate.

```
char* lpC = 0;
*lpC = 'A';
```

The code declares a character pointer and sets it to point at address 0. In most operating systems, including Windows CE, address 0 is protected and cannot be used. The second line attempts to place a character into the address pointed to by `lpC`, and since the address is protected, a page fault is generated and the application will fail.

Surprisingly, if you attempt to run these two lines of code in Windows CE you *will not* get a page protection fault—the application will continue to execute, although it may not function correctly. This can be a real problem in application development. To ensure that your page faults are correctly reported you will need to use exception handling.

Exception handling allows you to execute code and to trap any errors that would normally be reported by the operating system. Exception handling is a long and complex topic, especially with regard to the rules of how exceptions are handled with C++ object creation and destruction and to nested function calls. To confuse the issue, three types of exception handling exist in Windows programming: MFC (Microsoft Foundation Class), C++ language, and Windows structured exception handling (SEH).

I use Windows structured exception handling (SEH) to trap address and memory exceptions in my applications, and I try to keep it as simple as possible. With SEH the code needed to trap errors is placed in a `__try` block (that is try with two leading underscores). Errors generated in any function called from this block of code will be trapped. The error-trapping code to be executed in event of an error is placed in an `__except` block. The `EXCEPTION_EXECUTE_HANDLER` constant in the `__except` block indicates that errors will be handled by the block and not passed to other handlers.

```
__try
{
  char* lpC = 0;
  *lpC = 'A';
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
  MessageBox(hWnd,
    _T("Page Fault Caught in exception handling!"),
    szTitle, MB_OK | MB_ICONEXCLAMATION);
}
```

Now, even in Windows CE, the assignment to a `NULL` pointer will be trapped and reported. When writing a Windows API function with a message-handling function for a main window, I generally place a `__try/__except` block around all the code in the message-handling function. Nearly all the code in the application will be called from this function, so any page fault in any function called from the message handler will be trapped.

```
LRESULT CALLBACK WndProc(HWND hWnd,
      UINT message, WPARAM wParam, LPARAM lParam)
{
  __try
  {
    switch (message)
    {
      case WM_CREATE:
        break;
```

```
        // ... standard message handling code here
        default:
          return DefWindowProc(hWnd,
            message, wParam, lParam);
      }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
      MessageBox(hWnd,
        _T("Page Fault in exception handling!"),
        szTitle, MB_OK | MB_ICONEXCLAMATION);
    }
    return 0;
  }
```

Conclusion

Now that the preliminaries—what the book is about, the sample code, and general programming techniques such as error trapping—have been dealt with, you are ready to find out about the great features provided by Windows CE programming, such as communications, databases, and components. You can read the book chapter by chapter or, if you like, dip into those chapters that are important for you and the applications you are building. Before you start, one last thought: Remember that nearly all errors in an application are your errors, and just a very few may be due to bugs in the Windows CE operating system.