Windows® CE 3.0

APPLICATION PROGRAMMING

- Windows CE 3.0 Programming for Pocket PC®, Handheld PC, and embedded devices
- Enterprise computing including COM, DCOM, database access using ADOCE, and Microsoft® Message Queue
- Communications, including Web access with HTTP, TCP/IP sockets, serial communications, and desktop synchronization with ActiveSync® 3
- Build and run applications in Visual C++® using Microsoft Foundation Classes
- CD-ROM with eMbedded Visual C++ 3.0 and Pocket PC SDK

Nick Grattan
Marshall Brain

PH PTR MICROSOFT® TECHNOLOGIES SERIES

Windows CE 3.0
Application Programming

ISBN 0-13-025592-0

PRENTICE HALL PTR MICROSOFT® TECHNOLOGIES SERIES

NETWORKING

- Microsoft Technology: Networking, Concepts, Tools
  Woodard, Gattuccio, Brain
- NT Network Programming Toolkit
  Murphy
- Building COM Applications with Internet Explorer
  Loveman
- Understanding DCOM
  Rubin, Brain
- Web Database Development for Windows Platforms
  Gutierrez

PROGRAMMING

- The Windows 2000 Device Driver Book, Second Edition
  Baker, Lozano
- WIN32 System Services: The Heart of Windows 98 and Windows 2000, Third Edition
  Brain, Reeves
- Programming the WIN32 API and UNIX System Services
  Merusi
- Windows CE 3.0: Application Programming
  Grattan, Brain
- The Visual Basic Style Guide
  Patrick
- Windows Shell Programming
  Seely
- Windows Installer Complete
  Easter
- Windows 2000 Web Applications Developer's Guide
  Yager
- Developing Windows Solutions with Office 2000 Components and VBA
  Aitken
- Multithreaded Programming with Win32
  Pham, Garg
- Developing Professional Applications for Windows 98 and NT Using MFC, Third Edition
  Brain, Lovette
- Introduction to Windows 98 Programming
  Murray, Pappas
- The COM and COM+ Programming Primer
  Gordon
- Understanding and Programming COM+: A Practical Guide to Windows 2000 DNA
  Oberg
- Distributed COM Application Development Using Visual C++ 6.0
  Maloney
- Distributed COM Application Development Using Visual Basic 6.0
  Maloney
- The Essence of COM, Third Edition
  Platt
- COM-CORBA Interoperability
  Geraghty, Joyce, Moriarty, Noone
- MFC Programming in C++ with the Standard Template Libraries
  Murray, Pappas
- Introduction to MFC Programming with Visual C++
  Jones
- Visual C++ Templates
  Murray, Pappas
- Visual Basic Object and Component Handbook
  Vogel
- Visual Basic 6: Error Coding and Layering
  Gill
- ADO Programming in Visual Basic 6
  Holzner
- Visual Basic 6: Design, Specification, and Objects
  Hollis
- ASP/MTS/ADSI Web Security
  Harrison

BACKOFFICE

- Designing Enterprise Solutions with Microsoft Technologies
  Kemp, Kemp, Goncalves
- Microsoft Site Server 3.0 Commerce Edition
  Libertone, Scoppa
- Building Microsoft SQL Server 7 Web Sites
  Byrne
- Optimizing SQL Server 7
  Schneider, Goncalves

ADMINISTRATION

- Microsoft SQL Server 2000
  Fields
- Windows 2000 Cluster Server Guidebook
  Libertone
- Windows 2000 Hardware and Disk Management
  Simmons

- Windows 2000 Server: Management and Control, Third Edition
  Spencer, Goncalves
- Creating Active Directory Infrastructures
  Simmons
- Windows 2000 Registry
  Sanna
- Configuring Windows 2000 Server
  Simmons
- Supporting Windows NT and 2000 Workstation and Server
  Mohr
- Zero Administration Kit for Windows
  McInerney
- Tuning and Sizing NT Server
  Aubley
- Windows NT 4.0 Server Security Guide
  Goncalves
- Windows NT Security
  McInerney

CERTIFICATION

- Core MCSE: Windows 2000 Edition
  Dell
- Core MCSE: Designing a Windows 2000 Directory Services Infrastructure
  Simmons
- Core MCSE
  Dell
- Core MCSE: Networking Essentials
  Keogh
- MCSE: Administering Microsoft SQL Server 7
  Byrne
- MCSE: Implementing and Supporting Microsoft Exchange Server 5.5
  Goncalves
- MCSE: Internetworking with Microsoft TCP/IP
  Ryvkin, Houde, Hoffman
- MCSE: Implementing and Supporting Microsoft Proxy Server 2.0
  Ryvkin, Hoffman
- MCSE: Implementing and Supporting Microsoft SNA Server 4.0
  Mariscal
- MCSE: Implementing and Supporting Microsoft Internet Information Server 4
  Dell
- MCSE: Implementing and Supporting Web Sites Using Microsoft Site Server 3
  Goncalves
- MCSE: Microsoft System Management Server 2
  Jewett
- MCSE: Implementing and Supporting Internet Explorer 5
  Dell
- Core MCSD: Designing and Implementing Desktop Applications with Microsoft Visual Basic 6
  Holzner
- Core MCSD: Designing and Implementing Distributed Applications with Microsoft Visual Basic 6
  Houlette, Klander
- MCSD: Planning and Implementing SQL Server 7
  Vacca
- MCSD: Designing and Implementing Web Sites with Microsoft FrontPage 98
  Karlins

PRENTICE HALL PTR MICROSOFT® TECHNOLOGIES SERIES

Windows CE 3.0

Application Programming

Nick Grattan
Marshall Brain

Prentice Hall PTR, Upper Saddle River, NJ 07458
www.phptr.com

Library of Congress Cataloging-in-Publication Data

Grattan, Nick.
Windows CE 3.0 : application programming / Nick Grattan, Marshall Brain.
p. cm. — (Prentice Hall series on Microsoft technologies)
ISBN 0-13-025592-0
1. Application software—Development. 2. Microsoft Windows (Computer file)
I. Brain, Marshall. II. Title. III. Series.

QA76.76.D47 G76 2001
005.4'469—dc21

00-063708

Editorial/Production Supervision: *G&S Typesetters*
Acquisitions Editor: *Mike Meehan*
Editorial Assistant: *Linda Ramagnano*
Cover Design Director: *Jerry Votta*
Cover Designer: *Anthony Gemmellaro*
Manufacturing Manager: *Alexis R. Heydt*
Series Design: *Gail Cocker-Bogusz*
Marketing Manager: *Debby van Dijk*
Art Director: *Gail Cocker-Bogusz*
Buyer: *Maura Zaldivar*
Project Coordinator: *Anne Trowbridge*

Prentice-Hall, Inc.
Upper Saddle River, New Jersey 07458

Prentice Hall books are widely used by corporations and government agencies for training, marketing, and resale. The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:

Corporate Sales Department,
Prentice Hall PTR
One Lake Street
Upper Saddle River, NJ 07458
Phone: 800-382-3419; FAX: 201-236-7141
E-mail (Internet): corpsales@prenhall.com

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-025592-0

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Pearson Education Asia P.T.E., Ltd.
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

To my parents, Bob and Mildred Grattan.
Thanks for everything.
NG.

CONTENTS

T H R E E

Object Store, Directory, and Network Operations

Windows CE uses the Object Store for storing files, databases, and the registry (see Chapter 4). The Object Store uses RAM. This is limited to 256 MB in Windows CE 3.0, and 16 MB in earlier versions. Other devices can be used to store files and database, including storage cards (such as Compact Flash memory cards) and disk drives. Windows CE can also connect to resources on the network, either through a dialup/serial communications Remote Access Services (RAS) connection or a network device such as a NE2000 PCMCIA network card.

Unlike Windows NT/98/2000, Windows CE does not use drive letters (for example, `"F:"`) for network connections or devices. Directories in the Object Store (for example, `"\Storage Card"`) represent storage devices. Network connections can be accessed directly through UNCs (Universal Naming Conventions) such as `"\\myserver\myshare\myfile.txt"`. Alternatively, a connection can be made using the remote name (the UNC) and a local name. The local name is added to the directory `"\network"`, which can then be used to access the network. So, for example, if a connection is made using the local name `"myresource"`, and the network resource contains the file `"myfile.txt"`, the file can be accessed through the name `"\network\myresource\myfile.txt"`. Windows CE does not support the concept of "current directory," so functions like `GetCurrentDirectory` are not implemented.

The object store is maintained in RAM, and so needs to be reliable in the event of system crashes and invalid memory pointers from devices and applications. The object store uses transactions to ensure that the contents of the store can be returned to a known, integral state when a device is restarted. Files and directories are just two kinds of objects that can be stored. Registry items and property database records are also objects. Each object (including files and directories) has a unique identifier called an "Object ID," or OID. While you can find the OID for a file or directory, it is not particularly useful. However, the OIDs are essential when dealing with property databases.

Windows CE gives you several functions that you can use to access information about the object store, individual directories (folders), and network resources. For example, you use these functions:

- To find the maximum size and free space in the Object Store and storage devices
- To create and remove directories
- To find files in directories

Windows CE contains a set of `WNet` functions that lets you find and connect to network drives and printers shared by other machines. With these functions you can:

- Enumerate all the domains on the network
- Enumerate all the machines in each domain
- Enumerate all the drives and printers on each machine
- Connect to any drive on the network
- Disconnect from any drive

All the connection options seen by a user in the Explorer are implemented using the `WNet` and related functions.

Getting Object Store Free Space

Determining the available free space in the Object Store or storage device is important before attempting to save large amounts of data, or for providing feedback to the user. Listing 3.1 shows how to obtain this information by calling `GetDiskFreeSpaceEx`.

Listing 3.1 *Displays free space in the object store*

```
void Listing3_1()
{
  ULARGE_INTEGER ulFree, ulTotalBytes, ulTotalFree;
  // specify root directory in Object Store
  if(GetDiskFreeSpaceEx(_T("\\"),
    &ulFree, &ulTotalBytes, &ulTotalFree))
  {
    cout  << _T("Bytes available to caller: ")
          << tab << ulFree.LowPart << tab
          << ulFree.HighPart << endl;
    cout  << _T("Total number bytes: ")
          << tab << ulTotalBytes.LowPart << tab
          << ulTotalBytes.HighPart << endl;
```

Conclusion

This chapter describes how to use property databases to store structured information in the object store or in storage cards. The manipulation of data in property databases is carried out through API calls and not Structured Query Language (SQL) statements. Chapter 16 shows how databases can be accessed using ADOCE (Active Data Objects for Windows CE) with SQL. The chapter also shows how to add, query, and delete registry keys and data values stored in those keys.

F I V E

Processes and Threads

An application can be thought of as the `.exe` in the object store. When the application is run, a *process* is created. Therefore, a process is an instance of the application. There can be multiple instances of an application running at any one time; however, in devices such as the Pocket PC, it is considered best practice to have only a single instance running at any one time.

Each process starts off with a single, or primary thread. This thread starts executing code at the entry point, which is typically `WinMain` or `main` in most applications. An application can create additional threads to perform background tasks or to wait for some operation to complete. Using additional threads means that the primary thread is always available to deal with user interaction and to repaint the application windows. Multiple threads can be used for tasks such as:

- Waiting for data to arrive through a serial port, socket, or other communication medium
- Performing background operations, such as calculations

Using multiple threads provides many advantages but also carries responsibilities. Unless threads are coordinated (or 'synchronized') to ensure that no two threads attempt to use the same resource at the same time, they may both end up waiting for the other to complete a task. The techniques available in Windows CE to synchronize threads are covered in Chapter 6.

Not all threads are equal—they can be assigned different priorities. Windows CE supports 255 different thread priorities. Of these thread priorities, 248 are used for real-time applications and the remainder for ordinary applications. It is important to use thread priorities responsibly so that threads are neither starved of processor time nor use the processor to the exclusion of other threads.

A process will terminate when the primary thread in that process terminates and the resources used by that application are freed up. Unlike some other operating systems such as UNIX, there is no parent/child relationship. So, when a process terminates, processes created by that process are not automatically terminated.

Windows CE processes share many characteristics with desktop processes running on Windows NT/98/2000, but there are some significant differences. The virtual memory address space is arranged differently. In Windows NT/98/2000, each process has its own 4-GB virtual address space that is protected from being accessed by other applications. Remember, these address spaces are *virtual,* which means that the memory addresses may or may not be backed by real, physical memory.

In Windows CE, the operating system creates a single 4-GB address space. Each process is allocated a 32-MB address space called a 'slot'. The process uses this address space to map all the Dynamic Link Libraries (DLLs) that it needs to run, as well as data, heap, and stack. Certain larger allocations, such as memory-mapped files, may use address space outside the slot. Chapter 12 describes memory management in more detail. You will see the term 'module' used to refer to both applications and Dynamic Link Libraries.

Creating a Process with CreateProcess

The function `CreateProcess` is the standard way of creating processes. The code in Listing 5.1 prompts the user for the filename of an application (for example, 'pword.exe' for Pocket Word) and then calls `CreateProcess` to run the application.

Listing 5.1 *Creates a process with CreateProcess*

```
void Listing5_1()
{
  TCHAR szApplication[MAX_PATH];
  PROCESS_INFORMATION pi;
  if(!GetTextResponse(_T("Enter Application to Run:"),
      szApplication, MAX_PATH))
    return;
  if(CreateProcess(szApplication,
        NULL, NULL, NULL, FALSE, 0,
        NULL, NULL, NULL, &pi) == 0)
    cout << _T("Cannot create process") << endl;
  else
  {
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
  }
}
```

The CreateProcess function (Table 5.1) takes ten parameters, of which only two are essential in Windows CE for creating a process.

Table 5.1 *CreateProcess—Creates a new process*

CreateProcess	
LPCWSTR pszImageName,	Name of application to run.
LPCWSTR pszCmdLine	Command line arguments for application.
LPSECURITY_ATTRIBUTES psaProcess	Not supported, pass as NULL.
LPSECURITY_ATTRIBUTES psaThread	Not supported, pass as NULL.
BOOL fInheritHandles	Not supported, pass as FALSE.
DWORD fdwCreate	Flags specifying how to launch the application. Only the following are commonly used in Windows CE: CREATE_NEW_CONSOLE—Create a new console (only supported on platforms supporting cmd.exe) CREATE_SUSPENDED—Create process, but do not start executing thread.
PVOID pvEnvironment	Not supported, pass as NULL.
LPWSTR pszCurDir	Not supported, pass as NULL.
LPSTARTUPINFO psiStartInfo	Not supported, pass as NULL.
LPPROCESS_INFORMATION pProcInfo	Pointer to a PROCESS_INFORMATION structure.
HANDLE Return Value	Kernel object handle, or zero on error.

It is possible to pass a NULL for pszImageName, in which case pszCmdLine should point at a string containing the application filename followed by the command line arguments. If the application file name is not fully qualified, Windows CE will search in the '\Windows' folder followed by the root '\' folder. Platform builders can add an additional OEM-dependent folder and a '\ceshell' directory to the search.

The last argument to CreateProcess is a PROCESS_INFORMATION structure in which four pieces of information about the new process are returned:

- hProcess—A kernel object handle for the new process
- hThread—A kernel object handle for the primary thread for the new process
- dwProcessId—The process's system-wide unique identifier
- dwThreadId—The thread's system-wide unique identifier

Kernel object handles and identifiers are described in the next section. For now, note that the handles returned in the PROCESS_INFORMATION structure must be closed by calling CloseHandle.

Process Kernel Object Handles and Identifiers

The kernel object handles for the thread and process refer to data managed by the operating system relating to the thread or process. The operating system manages a reference count on the data—whenever a handle is returned to an application (as is the case with CreateProcess) or the handle is copied, the reference count is incremented. When an application is finished with the handle, it must call CloseHandle for that handle. The reference count is decremented when the handle is closed.

The lifetime of the kernel object is not necessarily the same as the lifetime of the process that it represents. If the reference count is greater than 0 when the process terminates, the kernel object will not be deleted. This means that information about the process can still be obtained even after the process has terminated. It is important that an application does call CloseHandle on the kernel object handle when the application is finished with the handle, to ensure that the operating system can free resources associated with the kernel object.

Process and thread kernel object handles are process-relative, that is, they can only be used reliably in the process that obtained them. Unlike Windows NT/98/2000, the function DuplicateHandle is not implemented in Windows CE, and so handles cannot be duplicated to allow them to be passed to other processes.

Some functions require a process or thread identifier rather than a kernel object handle. These identifiers are DWORD values that are unique for the process or thread across the entire operating system and can therefore be safely passed from process to process. The function OpenProcess may be used to obtain a process handle from a process identifier:

```
HANDLE hProcess;
hProcess = OpenProcess(0, FALSE, dwProcessId);
```

In Windows CE the first two parameters to OpenProcess are not supported and should be passed as 0 and 'FALSE'. As usual, the handle returned from OpenProcess should be closed by passing it to CloseHandle.

A process can determine its own process identifier by calling the function GetCurrentProcessId—the function takes no arguments and returns a DWORD.

```
DWORD dwProcessId;
dwProcessId = GetCurrentProcessId();
```

The function GetCurrentProcess can be called to return a kernel object handle for the current process. You need to be careful when using this handle, as it is actually a 'pseudohandle'. The returned handle always refers to the current process, so if you pass the handle to another process, the handle refers to that second process. Note that you do not need to call CloseHandle on pseudohandles.

Creating a Process with ShellExecuteEx

On Windows CE, CreateProcess has limited functionality since most of the parameters are not supported. In particular, a STARTUPINFO structure cannot be passed to the function, so you do not have much control on how the process is created. However, you can use the shell function ShellExecuteEx to start a process, as long as your Windows CE platform supports the standard shell. The function can be used to open documents. For example, you may specify that 'mydocument.pwd' should be started, and ShellExecuteEx will automatically launch Pocket Word and open the document. Further, different verbs can be applied to the document, such as 'open', 'print', and so on so that different operations can be performed on the document. Finally, you can specify how the application will be displayed using any of the constants supported by the ShowWindow function.

The ShellExecuteEx is passed a pointer to a SHELLEXECUTEINFO structure which is initialized to contain the required options for launching the application. In Listing 5.2 the user is prompted for an application or document to open, and the SHELLEXECUTEINFO structure is initialized and passed to ShellExecuteEx.

Listing 5.2 *Creates a process with ShellExecuteEx*

```
void Listing5_2(HWND hWnd)
{
  SHELLEXECUTEINFO sei;
  TCHAR szApplication[MAX_PATH];

  if(!GetTextResponse(_T("Enter Application to Run:"),
        szApplication, MAX_PATH))
    return;

  memset(&sei, 0, sizeof(sei));
  sei.cbSize = sizeof(sei);
  sei.hwnd = hWnd;
  sei.lpVerb = _T("open");
  sei.lpFile = szApplication;
  sei.lpParameters = NULL;
```

```
  sei.nShow = SW_SHOWNORMAL;
  if(ShellExecuteEx(&sei) == 0)
    cout  << _T("Error calling ShellExecuteEx:")
          << GetLastError() << endl;
}
```

Many of the members in the SHELLEXECUTEINFO structure are unused in Windows CE. Further, an instance handle to the new application should be returned, but this is not the case on all platforms. Table 5.2 lists the relevant members.

Table 5.2 *SHELLEXECUTEINFO—Relevant structure members*

Structure Member	Purpose
DWORD cbSize	Size of the structure in bytes.
HWND hwnd	Handle of a window to act as a parent for any dialogs shown by ShellExecuteEx.
LPCSTR lpVerb	Verb to use, for example, 'open', 'print', 'edit'.
LPCSTR lpFile	Executable file or document name.
LPCSTR lpParameters	Parameters to be passed to an application.
int nShow	How to display the application. Constants such as SW_HIDE and SW_SHOWNORMAL from the ShowWindow function can be used.

Waiting for a Process to Terminate

In many situations an application will start another application to perform some task (such as processing a file or connecting to a network), and will need to wait until the second application has completed the task. Further, the application will need to determine if the second application completed the task successfully or not. This can be achieved by calling the WaitForSingleObject function and using the process's exit code.

Process kernel objects can be in one of two states—signaled and nonsignaled. A process kernel object that represents a running process is nonsignaled. The process kernel object becomes signaled when the process terminates. The WaitForSingleObject (Table 5.3) can be passed a process kernel object, and the function call will block (that is, not return) until the process kernel object becomes signaled (which happens when the process itself terminates).

The code in Listing 5.3 creates a process and then calls WaitForSingleObject, passing in the process kernel object handle and the amount of time

Table 5.3 *WaitForSingleObject—Waits for a kernel object to be signaled*

`WaitForSingleObject`	
`HANDLE hHandle`	Kernel Object Handle to wait to become signaled.
`DWORD dwMilliseconds`	Number of milliseconds to wait before timing out, or `INFINITE` for no timeout.
`DWORD Return Value`	Return value: `WAIT_TIMEOUT`—Timeout value was exceeded. `WAIT_OBJECT_0`—Kernel object became signaled. `WAIT_FAILED`—Failure in function call, for example, handle is invalid.

to wait. (In this case, `INFINITE` causes `WaitForSingleObject` to block until the process terminates, regardless of how long this may be.) The call to `WaitForSingleObject` will not return until the process started by `CreateProcess` terminates. It is important that `CloseHandle` is called after `WaitForSingleObject`, otherwise the call will fail since the kernel object handle is invalid.

Listing 5.3 *Waits for a process to terminate*

```
void Listing5_3()
{
  TCHAR szApplication[MAX_PATH];
  PROCESS_INFORMATION pi;

  if(!GetTextResponse(_T("Enter Application to Run:"),
        szApplication, MAX_PATH))
    return;
  if(CreateProcess(szApplication,
        NULL, NULL, NULL, FALSE, 0,
        NULL, NULL, NULL, &pi) == 0)
    cout << _T("Cannot create process") << endl;
  else
  {
    if(WaitForSingleObject(pi.hProcess,
        INFINITE) == WAIT_FAILED)
      cout << _T("Could not wait on object");
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
  }
}
```

Calling `WaitForSingleObject` is the most efficient way of waiting for a process to terminate. The call does not consume processor time while it is waiting, and does not stop the power management functions of the operating

system. `WaitForSingleObject` is one of the synchronization functions supported by Windows CE and is described in more detail in Chapter 6.

Process Exit Code

Each process has an exit code that can be accessed by other processes and can be used to indicate success or failure. A process sets an exit code in the value returned from either WinMain or the 'C' main function. A process could, for example, return 0 to indicate success or a non-zero value indicating an error code.

An application can call the function `GetExitCodeProcess` to obtain the exit code for another application. This function is passed the process kernel object handle to the other process and a pointer to a `DWORD` to receive the exit code:

```
DWORD dwExitCode;
GetExitCodeProcess(hProcess, &dwExitCode);
```

On return, `dwExitCode` will contain the exit code set by the application, or the value `STILL_ACTIVE` if the application has not yet terminated. So, you could call `GetExitCodeProcess` after `WaitForSingleObject` returns in Listing 5.3.

In Windows NT/98/2000 the `ExitProcess` function can be called at any time to terminate the application, and this function is passed the exit code for the application. However, this function is not available in Windows CE. You can call `PostQuitMessage` instead, but this will eventually terminate the message loop and then exit `WinMain`, and in doing so, set an exit code.

An application can terminate another application through calling the function `TerminateProcess`, and can pass a value to be used as the exit code for the application being terminated. However, as described in the section "Terminating a Process," it is best not to terminate applications using this function.

Listing Running Processes

Windows CE provides a subset of the 'toolhelp' functions that provide information on, among other things:

- The processes running on the device
- The threads owned by each process
- The modules (for example, Dynamic Link Libraries) loaded by an application

These functions work by first creating a snapshot of the required information (for example, the list of processes). Calling the function `CreateTool-`

help32Snapshot does this. Then, the appropriate enumeration functions are called to obtain information about each object in turn. For processes, the functions are Process32First and Process32Next. Listing 5.4 shows code for listing all processes running on a device, together with the number of threads and process identifier. You need to include the file 'tlhelp32.h' and add toolhelp.lib when using these functions.

Listing 5.4 *Lists running processes*

```
#include <Tlhelp32.h>
// Add toolhelp.lib to the project

void Listing5_4()
{
  HANDLE          hProcessSnap;
  PROCESSENTRY32 pe32;

  // Take a snapshot of all processes currently running.
  hProcessSnap =
    CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if (hProcessSnap == (HANDLE)-1)
  {
    cout  << _T("Could not take Toolhelp snapshot")
          << endl;
    return ;
  }

  pe32.dwSize = sizeof(PROCESSENTRY32);

  if (Process32First(hProcessSnap, &pe32))
  {
      do
      {
        cout << pe32.szExeFile
             << _T(" Threads: ") << pe32.cntThreads
             << _T(" ProcID: ") << pe32.th32ProcessID
             << endl;
      }
      while (Process32Next(hProcessSnap, &pe32));
    }

    CloseToolhelp32Snapshot(hProcessSnap);
    return ;
}
```

The function CreateToolhelp32Snapshot is passed a constant for the first parameter that defines the information to be included in the snapshot. In this case TH32CS_SNAPPROCESS specifies that a snapshot of processes be produced. This function can also be used to create snapshots of the heap list (TH32CS_SNAPHEAPLIST), the modules being used by a process (TH32CS_

SNAPMODULE), and the threads for a process (TH32CS_SNAPTHREAD). These constants can be combined to create a snapshot that contains several different objects, or TH32CS_SNAPALL can be used to specify that all objects should be included. The second argument in CreateToolhelp32Snapshot specifies the process identifier of the process to be included in the snapshot—in this case '0' indicates all processes.

A handle is returned from CreateToolhelp32Snapshot that is used when enumerating the objects.

The function Process32First returns information about the first process in the snapshot. The function is passed the handle returned from CreateToolhelp32Snapshot, and a pointer to a PROCESSENTRY32 structure into which the information is placed. Note that the dwSize member of PROCESSENTRY32 must first be initialized with the size of the structure. The function Process32Next is called to obtain information about the next process—A return of TRUE indicates that another process's information was copied into the structure, and FALSE indicates the enumeration is complete.

Table 5.4 *PROCESSENTRY32 members returned in Windows CE*

Member	Purpose
DWORD dwSize	Size of the structure in bytes. Set before passing to functions.
DWORD th32ProcessID	Process Identifier. May be passed to other process functions such as TerminateProcess.
DWORD cntThreads	Number of threads owned by the process.
TCHAR szExeFile[MAX_PATH]	Null terminated string containing the path and name of the executable.
DWORD th32MemoryBase	Memory address of where the executable is loaded in the address space.

Table 5.4 shows the PROCESSENTRY32 members used in Windows CE and their purpose.

Modules Used by a Process

The toolhelp functions can also return a list of modules (normally DLLs) used by the application. To obtain the snapshot, CreateToolhelp32Snapshot is passed the TH32CS_SNAPMODULE constant, and the second parameter contains the process identifier whose module list is to be returned. In Listing 5.5 the process identifier for the current process is returned from calling GetCurrentProcessId. The functions Module32First and Module32Next are used to

enumerate the modules, and information about the modules is returned in a MODULEENTRY32 structure.

Listing 5.5 *Lists modules being used by a process*

```
void Listing5_5()
{
  HANDLE hModuleSnap;
  MODULEENTRY32 me32;
  DWORD dwProcessID;

  dwProcessID = GetCurrentProcessId();

  hModuleSnap =
      CreateToolhelp32Snapshot(TH32CS_SNAPMODULE,
        dwProcessID);

  if (hModuleSnap == (HANDLE)-1)
  {
      cout  << _T("Could not take Toolhelp snapshot")
            << endl;
      return ;
  }

  me32.dwSize = sizeof(MODULEENTRY32);

  if (Module32First(hModuleSnap, &me32))
  {
    do
    {
      cout << me32.szModule
           << _T(" Base addr: ")
           << (DWORD)me32.modBaseAddr
           << _T(" Size (KB): ")
           << me32.modBaseSize / 1024 << endl;
    }
   while (Module32Next(hModuleSnap, &me32));
  }
  CloseToolhelp32Snapshot (hModuleSnap);
  return;
}
```

In Listing 5.5 the name of the module (szModule) is displayed together with the base address (modBaseAddr) at which the module is mapped, and the size of the address space (modBaseSize) used by the module. DLLs are loaded at the top of the process's 32-MB slot. The value returned in modBaseSize is the size of the virtual address space used by the module and is not the amount of RAM used by the module. For example, a DLL could be mapped from ROM with the code being executed in place.

Table 5.5 *MODULEENTRY32 members returned in Windows CE*

Member	Purpose
`DWORD dwSize;`	Size of the structure in bytes. Set before passing to functions.
`DWORD th32ProcessID`	Process identified for the process being inspected.
`DWORD GlblcntUsage`	Number of times this module has been loaded in all applications running on the device.
`DWORD ProccntUsage`	Number of times this module has been loaded in the context of this process.
`BYTE *modBaseAddr`	Memory address where the module is loaded in the process's address space.
`DWORD modBaseSize`	Number of bytes of address space used in the mapping.
`HMODULE hModule;`	Handle to the module.
`TCHAR szModule [MAX_MODULE_ NAME32 + 1]`	Name of the module, not qualified with a path name.

The `GlblcntUsage` member contains the number of times this module has been loaded in all processes running on the device. The `ProccntUsage` value is the number of times the module has been loaded in the process being inspected. This can be larger than 1 since the application as well as other modules may reference the module in question.

Windows CE does not return `szExePath` member—in Windows NT/98/2000 this contains the fully qualified pathname of the module. The `hModule` member of `MODULEENTRY32` can be passed to the `GetModuleFileName` function, and this returns a fully qualified filename.

```
TCHAR szPathname[MAX_PATH];
GetModuleFileName(me32.hModule, szPathname, MAX_PATH);
```

The function `GetModuleFileName` is passed the handle to the module, a pointer to a character buffer to receive the fully qualified filename, and the maximum number of characters that can be placed in the buffer.

Terminating a Process

There are rare occasions when you will need to terminate another process from your application. Calling `TerminateProcess` does this.

```
DWORD dwExitCode = 1;
if(TerminateProcess(hProcess, dwExitCode))
  cout << _T("Process Terminated");
```

The function `TerminateProcess` is passed the handle of the process to terminate as the first parameter and a `DWORD` containing the exit code to use

for the process. The process being terminated does not have the opportunity to set an exit code, so one must be provided.

You should avoid calling `TerminateProcess`, since DLLs being used by the process do not have `DllMain` called with the reason code `DLL_PROCESS_DETACH`. Therefore, the DLLs cannot free resources they are using and resource or memory leaks can result.

Determining If a Previous Instance of a Process Is Running

Makers of certain target devices, such as Pocket PC, recommend that only a single instance of your application run at any time. In the event the user attempts to start a second instance, the second instance should terminate and bring the first instance to the foreground. The code in Listing 5.6 shows how to use `FindWindow` to locate the main application window of the first instance. The function `FindWindow` is passed the class name of the window. The second parameter is the window title, which is passed as `NULL` so that any window title will result in a match.

If `FindWindow` returns `NULL`, then this is the first instance of the application, and the application can continue executing. A non-`NULL` value means that another instance is running, so the current instance calls `ShowWindow` (to ensure the main application window of the first instance is not hidden) and `SetForegroundWindow` (to bring the main application window of the first instance to the foreground). This activates the thread and gives the window the input focus. Finally, the current instance terminates itself by calling `PostQuitMessage`. This code would be executed as soon as the application starts, and before any windows have been created.

Listing 5.6 *Handles second instances of an application*

```
BOOL Listing5_6()
{
  HWND hWnd;

  hWnd = FindWindow(_T("EXAMPLES"), NULL);
  if(hWnd == NULL)
    return FALSE;        // this is the first instance
  ShowWindow(hWnd, SW_SHOWNORMAL);
  SetForegroundWindow(hWnd);
  PostQuitMessage(0);    // terminate this instance
  return TRUE;
}
```

The second instance may need to pass information to the first instance, for example, specifying a document to open. This requires some form of interprocess communication. The simplest approach is for the second instance to send a message containing the data to the first instance. Alternatively, more sophisti-

cated interprocess communication techniques can be used, such as memory-mapped files. This requires that access to the shared memory be synchronized, which is covered in the next chapter.

Threads

Threads execute code. Each process starts out with a single primary thread that executes the entry point function (usually `WinMain` or the 'C' `main` function). This thread can create secondary threads by calling the `CreateThread` function. Through thread scheduling, multiple threads appear to execute simultaneously. Only one thread can actually be running at a time, so the operating system gives each thread a small amount of processor time (called a quantum) based on a scheduling algorithm based on thread priorities.

Threads are created by applications to

- Wait for some event to occur, such as termination of a process or receipt of information through a communications channel
- Perform background processing, such as calculations or database querying

Additional threads are usually created to wait for an event to occur so the primary thread is not blocked (that is, waiting for an event to occur). If the primary thread is blocked, the application will not be able to redraw the user interface or respond to user input. In Listing 5.3 the primary thread was blocked through calling `WaitForSingleObject`, and the application would be unresponsive until the application started with `CreateProcess` terminates. This code could be improved by creating a secondary thread, and calling `WaitForSingleObject` on that thread. Techniques for doing this are described in the next sections of this chapter.

As it happens, most secondary threads are blocked waiting for an event to occur. They are not, therefore, consuming processor time. There are a few occasions when secondary threads are used to perform background processing. In this situation, a thread will be using processor time. Windows CE will not enter a power-saving state when a thread is executing, so care needs to be taken to ensure that such threads do not execute for too long.

Thread synchronization becomes an issue whenever you have more than two threads in an application. Synchronization techniques ensure that threads access shared resources in an ordered way and allow threads to communicate information to each other. Chapter 6 looks at synchronization techniques.

User-Interface and Worker Threads

There are two types of threads:

- User-interface threads
- Worker threads

A user-interface thread is capable of handling messages, so a user-interface thread can create windows. Each user-interface thread must have its own message loop to handle messages for windows created by that thread. An application's primary thread is a user-interface thread, and it will typically have a message loop.

A worker thread does not have a message loop and therefore cannot create windows. The thread can, however, send messages to a window handle created by a user-interface thread, and can display a message box using the `MessageBox` function.

While it is possible to have multiple user-interface threads in an application, it is rarely absolutely necessary. To keep application design simple, you should execute all user-interface code with the primary thread, and create secondary worker threads for any task that would, if executed on the primary thread, make the application unresponsive.

Threads do require memory and other resources and do take time to create. Therefore, you want to limit the number of threads that your application creates. Further, many threads can make your application design much more complex and introduce synchronization problems.

Accessing Global and Local Variables In Threads

Global or static variables are accessible by all threads in an application. Auto, or function-local variables, are placed on the thread's stack, and are therefore only accessible by the thread that calls the function. Synchronization techniques must be applied whenever more than one thread accesses a global variable. Synchronization techniques are not generally required when accessing auto variables on the stack.

Each thread has its own stack on which variables local to a function are placed. The maximum stack size in Windows CE is 58 KB. When a thread is created, Windows CE reserves a 60-KB region in the process's virtual address space and commits memory as the stack grows. If the stack cannot be grown (because of lack of physical memory), the thread will be suspended until the request can be granted. The number of threads that can be created is limited by the amount of free virtual memory address space in the process and the available physical memory.

Using Correct Thread Processing

You should be sure to write code that does not interfere with the Windows CE thread schedule. If you do write such code, you can

- Take up valuable processor time
- Stop Windows CE from using power-saving techniques

A common mistake is to write a 'while' loop to wait until some event (such as termination of a process) has completed, as shown in this pseudocode:

```
while( ProcessHasNotFinished())
{
  ; // do nothing
}
```

In fact, this while loop does lots! It repeatedly calls the function ProcessHasNotFinished, and has scheduled processor time to do this. While a thread is executing code Windows CE cannot enter into one of its power-saving states, and so battery power will be wasted. You can improve this code somewhat by putting the thread to sleep for a short while on each loop. This will reduce the amount of processor time taken up.

```
while( ProcessHasNotFinished())
{
  Sleep(100); // suspend thread for 100 milliseconds
}
```

However, the best solution is to use a function like WaitForSingleObject to block the thread until the process terminates. Then, the thread takes up no processor time and does not interfere with battery-saving routines.

Some desktop applications make use of idle time to do background processing using the primary thread. Modifying the standard message loop to use PeekMessage, as shown in the following pseudocode, does this.

```
while(TRUE)
{
  if(PeekMessage(...))
    {
    // Call GetMessage, translate and
    // dispatch message.
    GetMessage(...);
    DispatchMessage(...);
    if(message is WM_QUIT)
      break;
    }
  else
    {
    // do we have background processing?
    if(bHaveBackgroundProcessing)
      DoBackgroundProcessing();
    }
}
```

This code, unfortunately, is not much better than the 'while' loop, and it will take up processing time. The background processing should be carried out in a thread.

Creating a Thread

Threads are created by calling the function `CreateThread`. The function is passed an address of a function (the 'thread function') that the new thread will start executing, in much the same way the primary thread starts executing `WinMain` as an entry point into the application. The thread function always has the following prototype:

```
DWORD WINAPI ThreadProc(LPVOID lpParameter);
```

The function is passed an `LPVOID` pointer that can be used for passing information into the thread from the thread that calls `CreateThread`. The function returns a `DWORD` which is the thread exit code. The thread exit code is used in much the same way as the process exit code described earlier in this chapter.

In Listing 5.7 `CreateThread` is called to create a new thread that starts executing the code in the thread function `MyThreadProc1`. The thread function displays a message and then returns. The thread terminates automatically on returning from the thread function in much the same way a process terminates when a return is made from `WinMain`.

Listing 5.7 *Creates a thread*

```
DWORD WINAPI MyThreadProc1(LPVOID lpParameter)
{
  cout << _T("Message from the thread") << endl;
  return 0;
}

void Listing5_7()
{
  HANDLE hThread;
  DWORD dwThreadId;

  hThread = CreateThread(NULL, 0, MyThreadProc1,
      NULL, 0, &dwThreadId);
  if(hThread == NULL)
    cout << _T("Could not create thread") << endl;
  else
  {
    CloseHandle(hThread);
    cout << _T("Thread Created") << endl;
  }
}
```

Table 5.6 shows the parameters for the `CreateThread` function. A thread has an identifier that is used when calling certain thread functions and is like

a process identifier. The function returns a kernel object handle that should be closed by calling `CloseHandle`.

Table 5.6 *CreateThread—Creates a new thread*

CreateThread	
`LPSECURITY_ATTRIBUTES lpThreadAttributes`	Ignored, pass as `NULL`
`DWORD dwStackSize`	Ignored, pass as `0`
`LPTHREAD_START_ROUTINE lpStartAddress`	Pointer to the thread function
`LPVOID lpParameter`	Pointer to data that is passed in to the `LPVOID lpParameter` parameter in the thread function
`DWORD dwCreationFlags`	`CREATE_SUSPENDED` to create the thread in a suspended state, or `0` to create a thread that is running
`LPDWORD lpThreadId`	`DWORD` pointer that receives the thread's identifier
`HANDLE Return Value`	Thread's kernel object handle, or `NULL` on failure

Unlike the Windows NT/98/2000 operating systems, in Windows CE a thread is always created with a default stack size of 58 KB. Note this is 58 KB of virtual address space, and physical memory is only allocated as the stack grows.

Terminating a Thread and Thread Exit Codes

A thread terminates when the thread function exits. The return value from the thread function is the thread's exit code. The exit code can be used to communicate success or failure to other threads. A thread function, or any function it calls, can terminate prematurely by calling `ExitThread`. This function is passed a single `DWORD` parameter that is used as the thread's exit code:

```
ExitThread(10); // set thread exit code to 10
```

The code in Listing 5.8 creates a thread, and then calls `WaitForSingleObject` to block the primary thread until this thread terminates, or until 5000 milliseconds have elapsed. It is a good idea to set a timeout on waiting for a thread just in case the thread function fails. The thread function calls `ExitThread` to prematurely terminate the thread and set the exit code to 10. Note that the output message and the return statement will never be executed. Once `WaitForSingleObject` unblocks the function, `GetExitCodeThread`

is called to retrieve the exit code (which should be 10). This function takes the thread handle and a pointer to a DWORD to receive the exit code.

Listing 5.8 *Thread exit codes*

```
DWORD WINAPI MyThreadProc2(LPVOID lpParameter)
{
  ExitThread(10);
  cout << _T("This message is not displayed") << endl;
  return 0;
}

void Listing5_8()
{
  HANDLE hThread;
  DWORD dwExitCode;

  hThread = CreateThread(NULL, 0, MyThreadProc2,
      NULL, 0, NULL);
  if(hThread == NULL)
    cout << _T("Could not create thread") << endl;
  else
  {
    if(WaitForSingleObject(hThread, 5000)
        == WAIT_FAILED)
      cout << _T("Could not wait on thread")
           << endl;
    GetExitCodeThread(hThread, &dwExitCode);
    CloseHandle(hThread);
    cout  << _T("Thread Exit code: ")
          << dwExitCode << endl;
  }
}
```

A thread can be terminated by another thread through calling TerminateThread. This function is passed the kernel object handle of the thread to terminate, and a DWORD specifying the exit code to set for the thread.

```
DWORD dwExitCode = 20;
if(!TerminateThread(hThread, dwExitCode))
  cout << _T("Could not terminate thread.") << endl;
```

As with TerminateProcess, you should only use TerminateThread as a last resort. Dynamic Link Libraries may have allocated thread local storage (TLS, described later in the chapter), and the DLLs will not have the opportunity to free this resource.

Thread States

A thread can exist in one of the following states:

- Suspended—Thread is not executing and will be suspended indefinitely.
- Running—Thread is executing code.
- Sleeping—Thread is sleeping for a specified period of time.
- Blocked—Thread is waiting for an event to occur, usually when calling `WaitForSingleObject`.
- Terminated—Thread has terminated, but the thread exit code is still available.

The functions `SuspendThread` and `ResumeThread` are used to change a thread from running to suspended, and vice versa. A thread can suspend itself but cannot resume itself since it is not executing, and so cannot call `ResumeThread`.

In Windows CE versions prior to 3.0, the minimum time a thread could be put to sleep was around 25 milliseconds. In version 3.0 the `Sleep` function can sleep a thread for a period of 1 millisecond or greater. This is due to changes to thread scheduling described later in this chapter. Further, the `GetTickCount` function (which returns the number of milliseconds elapsed since the device was powered-on) provides a resolution down to a millisecond. In Listing 5.9 the primary thread is put to sleep for a single millisecond, and the amount of time spent sleeping is recorded. In Windows CE 3.0, the program will record that the thread was asleep for around two milliseconds (the sleep time, plus overhead of calling `GetTickCount`), whereas in Windows CE versions prior to 3.0, a value of around 25 milliseconds or more will be recorded.

Listing 5.9 *Sleeps a thread*

```
void Listing5_9()
{
  DWORD dwTickCount = GetTickCount();
  Sleep(1);
  dwTickCount = GetTickCount()- dwTickCount;
  cout << _T("Sleep for: ") << dwTickCount << endl;
}
```

You can pass the value '0' to the `Sleep` function, and this yields control back to the thread scheduler regardless of whether the thread's time quantum was up. This can be used to allow other threads with the same priority an opportunity to execute immediately. This can be useful when synchronizing threads.

Thread Scheduling

The Windows CE thread scheduler is responsible for ensuring that threads get the proper amount of time to execute their code. Each thread is given a 'quantum' of time in which to execute code. Once the quantum of time has elapsed, the thread scheduler allows another thread to execute for its quantum. In Windows CE versions prior to 3.0, the quantum was set at 25 milliseconds. In Windows CE 3.0 it is set to 100 milliseconds (although this figure can be changed by an OEM). This means that in Windows CE 3.0 a thread can execute for up to 100 milliseconds without interruption.

A thread can change its quantum to make it longer or shorter. The code in Listing 5.10 displays the current quantum time for the thread using `CeGetThreadQuantum`. This function is passed the thread's handle that is obtained through calling `GetCurrentThread`. The `CeSetThreadQuantum` function is then called to set the quantum time for the current thread to 20 milliseconds.

Listing 5.10 *Thread quantums*

```
void Listing5_10()
{
  cout << CeGetThreadQuantum(GetCurrentThread()) << endl;
  CeSetThreadQuantum(GetCurrentThread(), 20);
}
```

The function `GetCurrentThread` used in Listing 5.10 returns a pseudohandle for the current thread. This handle does not have to be closed through a call to `CloseHandle` since it is a pseudohandle.

You might want to increase the thread quantum time if, for example, you are sending data to an instrument that needs to receive the data without interruption. You would set the quantum time to the period of time you expected the transmission to take. Then, once the transmission is complete, you would call return the quantum value back to its previous value, and then call `Sleep(0)` to end your quantum. Of course, you do not want to set the quantum period to be too long, otherwise operating system and other processes won't get an opportunity to execute. Finally, thread scheduling is dependent on the thread's priority, and this is discussed in the next section.

When the scheduler swaps out a thread, it saves a 'thread context'. This context contains the current state of the processor (including all the registers, program counter, stack frames, and so on). Then, when the thread is to receive its next quantum, the thread context is restored back into the processor and the thread set to running again. A thread can get another thread's context by calling the `GetThreadContext` function, passing the handle to the thread and a `CONTEXT` structure pointer. This is typically a debugging operation. The function `SetThreadContext` allows the current context for a thread to be changed.

The modified context will be used the next time the thread is scheduled for a quantum. The CONTEXT structure is highly dependent on the device's processor, since it contains members for CPU registers and so on. It is declared in the header file winnt.h.

Thread Priorities

The previous section described how threads are scheduled for execution using the quantum time period. The thread scheduler uses a round-robin algorithm for scheduling threads. However, this ignores the fact that threads can have different priorities, and this affects how frequently a thread is scheduled.

In Windows CE 3.0 a thread can be assigned any one of 255 different priorities, with 0 being the highest priority and 255 the lowest. The seven lowest priorities (255 to 249) are application thread priorities, while the remainder are real-time priorities. In Windows NT/98/2000 a thread has a priority relative to the process's priority class. Windows CE does not use priority classes, and each thread has a priority in its own right.

The scheduler first schedules threads at the highest priority level in a round-robin manner. Only when all threads at the highest level have blocked does the scheduler then schedule threads at the next highest level. This process is repeated down all the different priority levels. If, while a lower-priority thread is executing, a higher-level thread unblocks, the lower-priority thread is stopped executing (even if it has not finished its quantum), and the higher-priority thread is scheduled. A real time priority thread cannot be preempted (that is, swapped out) except by an interrupt-service routine even if its time quantum period has elapsed.

All threads are initially created at the 'normal' priority. The application then changes the thread's priority appropriately. A thread's priority can be set by either calling SetThreadPriority (to set an application priority) or CeSetThreadPriority (to set an application or real-time priority). The SetThreadPriority function is passed the handle to a thread and a constant specifying which priority to use (Table 5.7). Note that THREAD_PRIORITY_TIME_CRITICAL is a real-time priority. The constants in Table 5.7 have the values 0 for THREAD_PRIORITY_TIME_CRITICAL to 7 for THREAD_PRIORITY_TIME_CRITICAL. You can see that these constants do not map to the priority values of 0 to 255 used by CeSetThreadPriority. For this reason, you should only use these constants with SetThreadPriority.

There is only one situation where a thread's priority is automatically changed by the operating system, and that is *priority inversion*. If a low-priority thread is using a resource that a high-priority thread is waiting on, the operating system temporarily boosts the lower-priority thread until it releases the resource required by the higher-priority thread. Unlike Windows NT/98/2000,

Table 5.7 *CeSetThreadPriority priority constants*

Constant	Purpose
THREAD_PRIORITY_TIME_CRITICAL	Indicates 3 points above normal priority
THREAD_PRIORITY_HIGHEST	Indicates 2 points above normal priority
THREAD_PRIORITY_ABOVE_NORMAL	Indicates 1 point above normal priority
THREAD_PRIORITY_NORMAL	Indicates normal priority
THREAD_PRIORITY_BELOW_NORMAL	Indicates 1 point below normal priority
THREAD_PRIORITY_LOWEST	Indicates 2 points below normal priority
THREAD_PRIORITY_ABOVE_IDLE	Indicates 3 points below normal priority
THREAD_PRIORITY_IDLE	Indicates 4 points below normal priority

Windows CE does not provide 'foreground boosting' whereby the application in the foreground has its thread priorities set to a value greater than other applications.

In Listing 5.11 the code obtains the current thread priority by calling `CeGetThreadPriority`. This will generally return the value '251'. This corresponds to `THREAD_PRIORITY_NORMAL` in Table 5.7. A call is then made to `CeSetThreadPriority` to set the thread's priority to 140. This is a real time priority, so subsequent code will be executed without interruption. After displaying the new thread priority, the thread's priority is set back to its original value.

Listing 5.11 *Sets real time thread priorities*

```
void Listing5_11()
{
  int nPri = CeGetThreadPriority(GetCurrentThread());

  cout << _T("Pri: ") << nPri << endl;
  CeSetThreadPriority(GetCurrentThread(), 140);
  cout << _T("New Pri: ")
       << CeGetThreadPriority(GetCurrentThread())
       << endl;
  CeSetThreadPriority(GetCurrentThread(), nPri);
}
```

You should take care when using real time priorities. An application can easily take over the processor and not let other applications, or essential parts of the operating system, run correctly. If you do need to create real time threads, ensure that they remain real time for the minimum required time or remain blocked for the majority of time.

Enumerating Threads

The toolhelp functions can be used to list all the threads running on a device. The code in Listing 5.12 takes a snapshot of all threads by calling `CreateToolhelp32Snapshot` and passing the `TH32CS_SNAPTHREAD` constant. The second parameter is a process identifier, and 0 specifies that threads for all processes should be enumerated. The functions `Thread32First` and `Thread32Next` are used to enumerate the threads in the snapshot, and data on each thread is placed in a `THREADENTRY32` structure. As usual, `CloseToolhelp32Snapshot` should be called to close the snapshot.

Listing 5.12 *Lists running threads*

```
void Listing5_12()
{
  HANDLE hThreadSnap;
  THREADENTRY32 th32;

  hThreadSnap =
    CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

  if (hThreadSnap == (HANDLE)-1)
  {
    cout  << _T("Could not take Toolhelp snapshot")
          << endl;
    return ;
  }

  th32.dwSize = sizeof(THREADENTRY32);

  if (Thread32First(hThreadSnap, &th32))
  {
    do
    {
      cout  << _T("ThreadID: ")
            << th32.th32ThreadID
            << _T(" ProcessID: ")
            << th32.th32OwnerProcessID
            << _T(" Priority: ")
            << th32.tpBasePri << endl;
    }
    while (Thread32Next(hThreadSnap, &th32));
  }
  CloseToolhelp32Snapshot(hThreadSnap);
  return;
}
```

There are really only three members of `THREADENTRY32` that provide useful information:

- th32ThreadID—The thread identifier for the thread
- th32OwnerProcessID—The identifier for the process in which the thread runs
- tpBasePri—The thread's priority as a value between 0 and 255

Running Listing5_12 on a Pocket PC shows that most threads run at priority 251 (THREAD_PRIORITY_NORMAL), others at 255 (THREAD_PRIORITY_IDLE) and 249 (THREAD_PRIORITY_HIGHEST), and around 10 running at real time priorities such as 109, 132, 118 120 and 126.

Determine Thread Execution Times

The ThreadTimes function can be used to determine the amount of time in milliseconds that a thread has been executing. This can be useful when monitoring an application's performance. Listing 10.6 in Chapter 10 ("The Remote API") shows an example of using this function. In this case, ThreadTimes is called from a DLL running on a Windows CE device, with the data being returned through a RAPI call to a desktop application.

Creating Threads with MFC

MFC provides the CWinThread class to provide support for creating threads, and is similar in many respects to CWinApp in the methods it supports. This class can be used to manage worker and user interface threads because, like CWinApp, it implements a message loop.

Although MFC provides a class to manage threads, you still need to create a global thread function that has the following prototype:

```
UINT MFCThreadProc(LPVOID lpParameter);
```

The MFC function AfxBeginThread can be called to create a worker thread using the thread function:

```
CWinThread* pThread =
  AfxBeginThread(MFCThreadProc, NULL);
```

The function AfxBeginThread is passed a pointer to the thread function and a pointer to data to pass to the LPVOID parameter (which is, in this case, NULL). AfxBeginThread returns a pointer to a CWinThread object through which the newly created thread can be managed. For example, you can suspend and resume the thread by calling CWinThread::SuspendThread and CWinThread::ResumeThread. You can ignore the return result from AfxBeginThread if you do not need to manage the thread, and you do not have to delete the CWinThread object.

Conclusion

In this chapter you have found out about processes and how they are created and terminated. The chapter also covered creating additional threads in your application. However, this is only half of the story. As soon as your application creates additional threads, you need to make your application 'thread safe'. This means that all access to global variables and resources must be protected by using synchronization techniques. This is the subject of the next chapter.

S I X

Thread Synchronization

The last chapter showed how processes could create additional threads to carry out background tasks or to wait for some event to occur. However, using threads is not as simple as creating a new thread and leaving it to execute. Since all threads running in an application share the same global resources and variables, there is always the chance that two threads will attempt to access the same resource at the same time. Such simultaneous access of a global resource may cause the program to fail. Because of the way the threads are scheduled, the problems caused by simultaneous access of a global resource will not occur every time the program is run. Typically such synchronization problems occur rarely enough to make tracking them down difficult but frequently enough to be annoying for the user.

There is only one sure way to avoid synchronization problems: build in and test thread synchronization techniques whenever you create additional threads. If you have difficulties in writing synchronization code, you are better off staying with a single-threaded application. You can then use other methods, such as timers or sending messages, in place of additional threads.

The Need for Synchronization

Thread synchronization is required when

an application is multithreaded and these threads attempt to use global variables and resources, or the threads need to wait until some event has completed before continuing execution.

First, let's look at why synchronization is required when multiple threads access a global variable. In the following code, a global floating-point variable is declared, and two threads try to perform different actions on that variable.

```
float g_fValue = 10.0;

void f1()      // called by thread 1
{
  g_fValue = g_fValue * g_fValue;
}

void f2()      // called by thread 2
{
  g_fValue = 3.0 + g_fValue;
}
```

It is easy to see that the value in 'g_fValue' can be either (10*10) + 3 = 103 or (10+3)*(10+3) = 169 after the two threads have finished executing, depending on whether function 'f1' or function 'f2' completes first. The order in which the two functions execute depends on how the threads were started and scheduled.

However, there is a much more worrisome potential outcome—the variable 'g_fValue' may contain a completely different value after the functions have completed. While we think of a statement like 'g_fValue += 10;' as being atomic (that is, it will execute in its entirety all in one go without interruption), the statement is actually compiled into a number of machine code operations.

```
g_fValue = g_fValue * g_fValue;
  fld      dword ptr [g_fValue (0041060c)]
  fmul     dword ptr [g_fValue (0041060c)]
  fst      dword ptr [g_fValue (0041060c)]
g_fValue = 3.0 + g_fValue;
  fadd     qword ptr
                [__real@8@4000c000000000000000 (0040c020)]
  fstp      dword ptr [g_fValue (0041060c)]
```

From this listing it becomes obvious that the first statement 'g_fValue = g_fValue * g_fValue' is compiled into three different op codes. The thread quantum could finish after the first op code has completed, and the second thread may then be scheduled to execute the statement 'g_fValue = 3.0 + g_fValue'. Therefore, the resulting computation would be 10*(10+3) = 130. This scenario would be a very rare event, but it *could* happen. Thread synchronization techniques should be employed to prevent it from *ever* happening.

A related problem arises when a thread must complete a number of related steps as an atomic unit without interruption from other threads. For example, if you write an application to create and maintain a linked list, a thread that inserts a new item in the linked list must create the new item, link the new

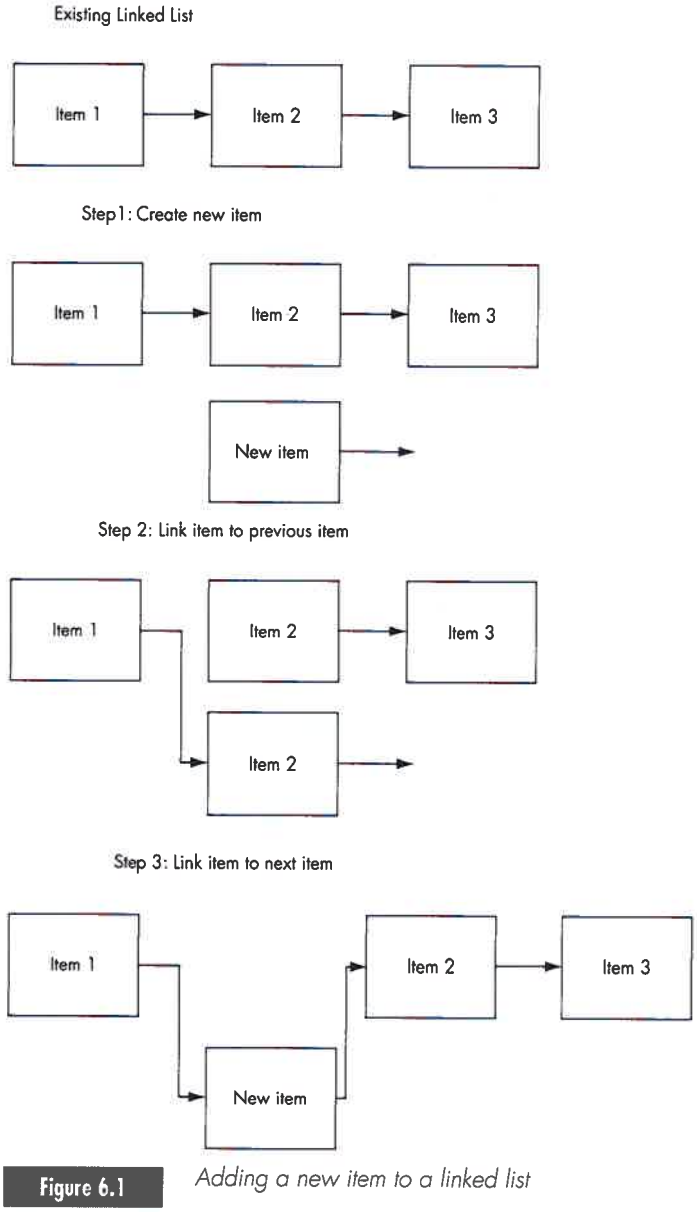
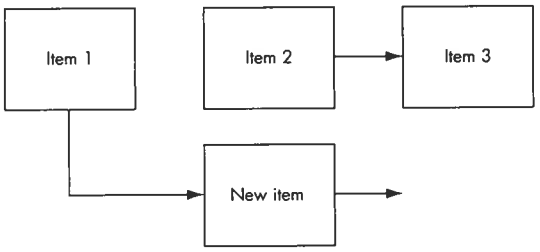


Figure 6.1 *Adding a new item to a linked list*

item to the previous item in the list, and link the new item to the next item in the list without other threads accessing the linked list (Figure 6.1).

If a second thread attempts to access the linked list before the new item has been linked to the next item in the list, the second thread will prematurely reach the end of the list when the new item is traversed (Figure 6.2).



If second thread starts accessing the list at Item1, it will stop when the 'new item' is accessed. Item 2 and 3 will be ignored.

Figure 6.2 *Threads add and access items at the same time*

Worse still, if two threads attempt to insert new items at the same point in the linked list, the list itself can be broken (Figure 6.3). This is because each thread is unaware of the links being created by the other thread. These are known as *race conditions* and require synchronization.

The second need for synchronization occurs when threads need to coordinate their executions based on some event being completed. In this situation one or more threads are typically blocked and are waiting for the event to occur. When two or more threads are waiting for two or more events to complete, there is a real chance that a 'deadlock' or 'deadly embrace' will occur. This should be avoided at all costs. Here is a typical situation that leads to a deadlock:

- Thread 1 has resource 1 locked and is blocked waiting on resource 2 to be freed.
- Thread 2 has resource 2 locked and is blocked waiting on resource 1 to be freed.

In this situation neither thread 1 nor thread 2 can continue executing because they are both blocked. Because the threads are blocked, the threads cannot execute code to free the resource they have locked (Figure 6.4). They therefore remain blocked forever. A deadlock between two worker threads is serious, but a deadlock between a worker thread and the primary thread is critical. The application will be not be responsive to the user, and the application will have to be closed down.

Synchronization techniques should be employed to ensure that threads block correctly, and perhaps provide timeouts to occur in the event of a deadlock. Deadlocks may occur infrequently in an application when a particular train of events occurs in a particular order. This makes them difficult to track down.

Deadlocks can be avoided by following this simple rule:

Always lock or block on a resource in the same order. All threads blocking or locking resource 1 and resource 2 should block or lock resource 1 before attempting to block or lock resource 2. Resources should be unlocked in the reverse order they were locked in.

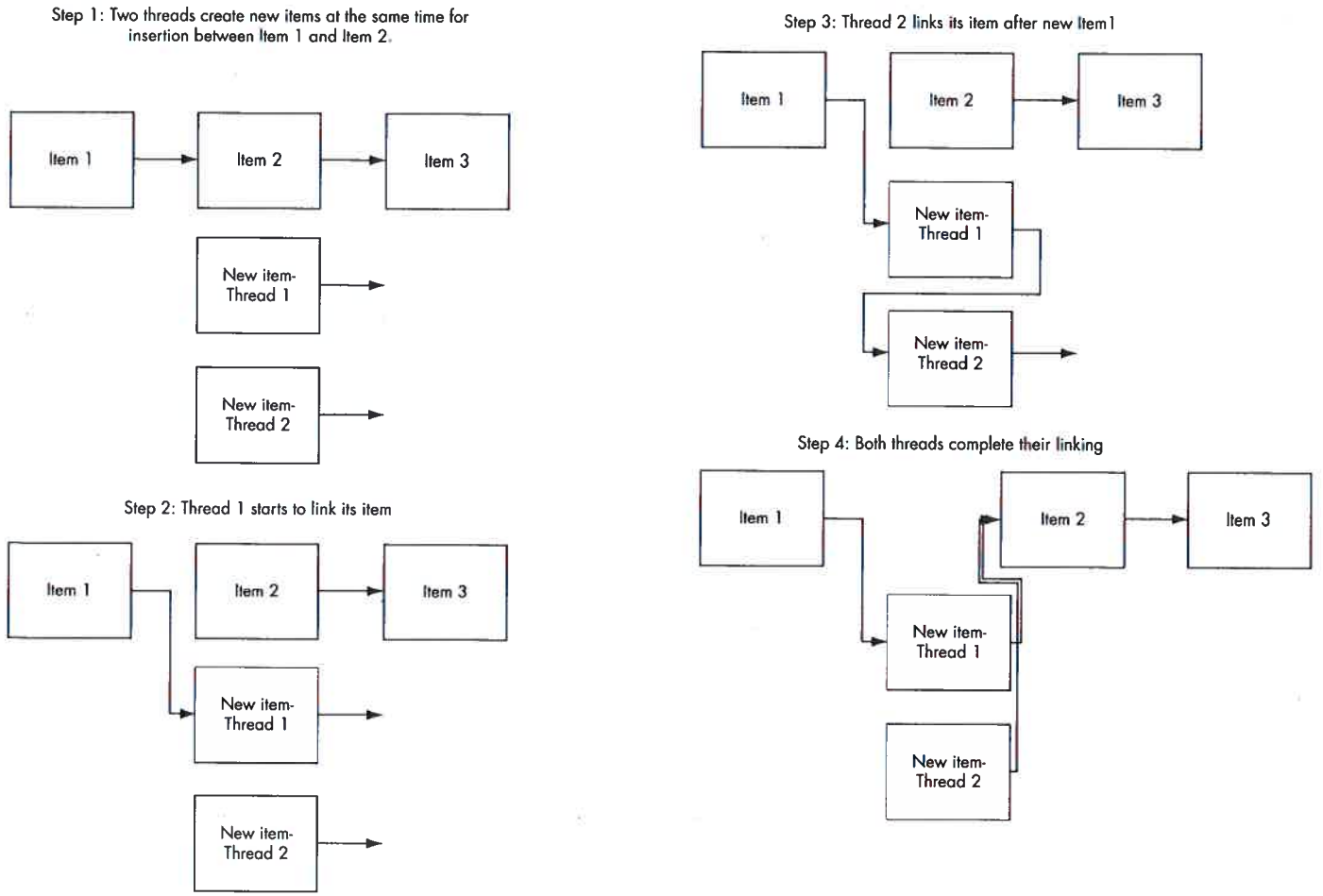


Figure 6.3 *Race conditions when two threads manipulate the linked list at the same time*

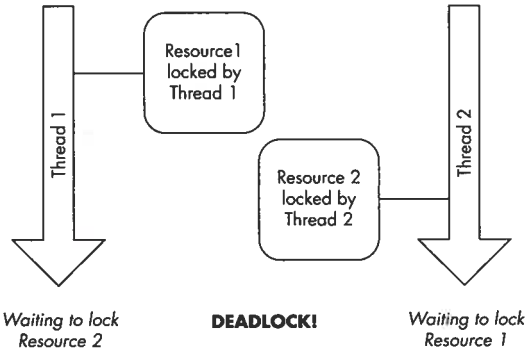


Figure 6.4 *Deadlock between two threads*

The scenario outlined above with thread 1 and thread 2 blocking on resource 1 and resource 2 leads to a deadlock because the resources were not locked in the same order. Applying this rule leads to the following:

- Thread 1 locks resource 1 and attempts to use resource 2. If resource 2 is not in use, thread 1 locks resource 2, uses the resources, and then unlocks resource 2 followed by resource 1.
- Thread 2 attempts to lock resource 1. If it is in use, thread 2 blocks. If it is not in use, thread 2 locks resource 1 and then attempts to lock resource 2. It will wait until resource 2 is available, use the resources, and then unlock resource 2 and then resource 1.

While this rule is quite simple, it can be difficult to implement if the code used to lock and block on the resources is scattered throughout the application. Therefore, you should write functions or classes that manage the locking or blocking.

One of the more difficult design issues is deciding which of the synchronization techniques available in Windows CE should be applied to your problem. After describing each of the techniques, the section "Selecting the Correct Synchronization Technique" later in the chapter provides a summary and a set of selection criteria.

Critical Sections

A critical section identifies code that must be executed to completion before another piece of code can be executed. In the example presented in the previous section, the statements '`g_fValue = g_fValue * g_fValue;`' and '`g_fValue = 3.0 + g_fValue;`' should be marked as critical sections to ensure that both statements can be executed to completion before the other starts executing. If this is done, the only two possible results in `g_fValue` are 103 and 169. The spurious value of 130 will never occur.

To create and use a critical section you should:

- Declare a CRITICAL_SECTION structure as a global variable, or a member variable of a class
- Call the InitializeCriticalSection function to initialize this structure
- Call EnterCriticalSection before the lines of code that form the critical sections
- Call LeaveCriticalSection after the lines of code that form the critical sections
- Call the DeleteCriticalSection function when the CRITICAL_SECTION is no longer required

All the critical section functions take a single argument that is a pointer to the CRITICAL_SECTION structure. You should treat this structure as a black box and not use the members contained in it. The code in Listing 6.1 declares a critical section structure g_cs, and creates two threads using thread functions f1 and f2. Each thread function performs an operation on a global float value 'g_fValue'. Because each thread is accessing a global function, the critical section structure g_cs is used to synchronize access to the global variable.

Listing 6.1 *Using critical sections*

```
float g_fValue = 10.0;
CRITICAL_SECTION g_cs;
DWORD WINAPI f1(LPVOID);
DWORD WINAPI f2(LPVOID);

void Listing6_1()
{
  HANDLE hThread1, hThread2;
  DWORD dwThreadID;

  g_fValue = 10.0;
  InitializeCriticalSection(&g_cs);
  hThread1 = CreateThread(NULL, 0,
      f1, NULL, 0, &dwThreadID);
  hThread2 = CreateThread(NULL, 0,
      f2, NULL, 0, &dwThreadID);
  // Wait until thread 1 and thread 2 completes
  WaitForSingleObject(hThread1, INFINITE);
  WaitForSingleObject(hThread2, INFINITE);
  DeleteCriticalSection(&g_cs);
  CloseHandle(hThread1);
  CloseHandle(hThread2);
  cout << _T("Finished:") << g_fValue << endl;
}
```

```
DWORD WINAPI f1(LPVOID)
{
  EnterCriticalSection(&g_cs);
  g_fValue = g_fValue * g_fValue;
  LeaveCriticalSection(&g_cs);
  return 0;
}

DWORD WINAPI f2(LPVOID)
{
  EnterCriticalSection(&g_cs);
  g_fValue = (float)3.0 + g_fValue;
  LeaveCriticalSection(&g_cs);
  return 0;
}
```

In Listing 6.1 you will notice that `WaitForSingleObject` is called twice, once for each of the two threads. This causes the function `Listing6_1` to block until both threads have terminated. This is important, since the call to `DeleteCriticalSection` cannot be made until both threads have finished using the critical section. `WaitForMultipleObjects` *cannot* be used for this purpose, since in Windows CE `WaitForMultipleObjects` only blocks until one of the threads terminates. This is described in more detail later.

Once one thread calls `EnterCriticalSection`, any other thread calling `EnterCriticalSection` using the same `CRITICAL_SECTION` structure will block until the first thread calls `LeaveCriticalSection`. When this happens, a thread blocked in `EnterCriticalSection` will unblock and can then execute the code in its critical section. Multiple threads can be blocked on calls to `EnterCriticalSection`, and you cannot predict which of these blocked threads will unblock. Note that creating a critical section does not ensure that the code in the critical section will execute to completion without interruption —the normal thread-scheduling rules apply.

The following rules should be applied when using critical sections:

- Always ensure that the `LeaveCriticalSection` call is made. For example, do not have 'return' statements in the critical section code.
- Do not introduce user interactions, such as a message box, in a critical section. Other threads will block until the user dismisses the message box.
- Do not have code that takes a long time to execute in a critical section. You will end up blocking other threads, and they won't be able to execute their code.

You can declare multiple `CRITICAL_SECTION` structures to protect, for example, the access to different global variables. While this can improve the multithreading processing (since threads will not unnecessarily be blocked), it introduces the potential of deadlocks. For example, a thread could enter critical section 1 and then attempt to enter critical section 2. Another thread could

enter critical section 2 and then attempt to enter critical section 1. If this happens simultaneously, a deadlock can occur. Using the rule described earlier, you can avoid this by always entering critical sections in the same order.

The Interlocked Functions

Ensuring protected access to global integer values turns out to be a common requirement in many applications, so the Windows CE API provides the 'interlocked' functions to allow safe incrementing, decrementing, and swapping of values in global integer variables. The functions are

- `InterlockedIncrement`—Increment an integer variable. The function takes a single parameter, which is a pointer to the integer to increment.
- `InterlockedDecrement`—Decrement an integer variable. The function takes a single parameter, which is a pointer to the integer to increment.
- `InterlockedExchange`—Places a new value into an integer variable. The first parameter is a pointer to an integer to receive the new value, and the second parameter is the new integer value.

For example, the following code declares a global integer value and uses `InterlockedIncrement` to increment the value in that variable. Using this function ensures that other functions using the interlocked functions will block until this call is completed.

```
LONG g_lMyVar;
InterlockedIncrement(&g_lMyVar);
```

Note that *all* changes to the variable `g_lMyVar` should be through the interlocked functions to ensure that correct synchronization occurs. If you need to perform a more complex calculation (for example, one that involves multiplication that does not have an interlocked function), you should perform the calculation using local variables, and then copy the value into the global integer variable using the `InterlockedExchange` function.

WaitForSingleObject and WaitForMultipleObjects

Synchronization relies on one thread blocking until another thread has completed a task that uses some sort of shared resource. In Windows CE two blocking functions are commonly used:

- `WaitForSingleObject`: Waits until a single kernel object becomes signaled, or a timeout occurs
- `WaitForMultipleObjects`: Waits until one of several kernel objects becomes signaled, or a timeout occurs

Chapter 5 ("Processes and Threads") showed how `WaitForSingleObject` could be used to block until a thread or process terminates. However, `WaitForSingleObject` can also be used to block on a wide range of synchronization objects, such as mutexes, events, and semaphores.

Table 6.1 *WaitForSingleObject—Blocks until object becomes signaled*

WaitForSingleObject	
`HANDLE hHandle`	Handle of kernel object to block on, for example, thread, process, mutex, event, or semaphore.
`DWORD dwMilliseconds`	Timeout value in milliseconds. The constant `INFINITE` specifies no timeout.
`DWORD Return Value`	`WAIT_OBJECT_0` if the object is signaled. `WAIT_TIMEOUT` if the wait timed out. `WAIT_ABANDONED` if a mutex object became abandoned (see section on mutex objects for abandoned mutex objects). `WAIT_FAILED` indicates failure, call `GetLastError` for detailed error information.

`WaitForSingleObject` can be called with a '0' value for `dwMilliseconds`. In this case, the function does not block but returns `WAIT_OBJECT_0` if the object is signaled, or `WAIT_TIMEOUT` if the object is not signaled. Calling the function in this way is used to determine if an object is signaled or non-signaled without blocking.

Table 6.2 *WaitForMultipleObjects—Blocks until first object becomes signaled*

WaitForMultipleObjects	
`DWORD nCount`	Number of kernel objects to wait on.
`HANDLE *lpHandles`	Array of kernel object handles to wait on.
`BOOL fWaitAll`	Must be `FALSE`. Windows CE does not support waiting on all object handles.
`DWORD dwMilliseconds`	Timeout value in milliseconds. The constant `INFINITE` specifies no timeout.
`DWORD Return Value`	`WAIT_OBJECT_0` to (`WAIT_OBJECT_0 + nCount -1`) indicating which object in the `lpHandles` array became signaled. `WAIT_ABANDONED_0` to (`WAIT_ABANDONED_0 + nCount -1`) indicating which event object was abandoned. `WAIT_TIMEOUT` if the wait timed out. `WAIT_FAILED` indicates failure, call `GetLastError` for detailed error information.

In Windows CE WaitForMultipleObjects will always return when the first kernel object becomes signaled, whereas in Windows NT/98/2000 WaitForMultipleObjects can be used to block until all the objects become signaled.

The array of object handles passed to WaitForMultipleObjects can include a mixture of different kernel objects, such as threads, processes, and so on. However, the same kernel object handle cannot appear more than once in the array.

Using Mutex Objects

Mutex (or 'Mutual Exclusion') kernel objects are used to ensure that global variables or resources are accessed exclusively by a piece of code. In this respect, they provide the same functionality as critical sections. However, they are more flexible. For example, critical sections can only be used to ensure exclusivity within a single process, whereas mutex objects can be used across processes.

The following steps are required when using mutex kernel objects:

- Create a new mutex or open an existing mutex by calling the function CreateMutex.
- Call WaitForSingleObject when entering critical code.
- Call ReleaseMutex when the critical code execution is complete.
- Call CloseHandle on the mutex when the mutex is no longer required.

Like all kernel objects, a mutex can either be signaled (in which case WaitForSingleObject will not block), or non-signaled (in which case WaitForSingleObject will block until the object becomes signaled). The function ReleaseMutex changes the mutex state from signaled to non-signaled.

Table 6.3 *CreateMutex—Creates a new mutex or opens an existing mutex*

CreateMutex	
LPSECURITY_ATTRIBUTES lpMutexAttributes	Not supported, pass as NULL.
BOOL bInitialOwner	TRUE if the object is created signaled, and will be owned by the thread creating the mutex. FALSE if the object is to be created non-signaled. The value is ignored if an existing mutex is being opened.
LPCTSTR lpName	String containing name of mutex, or NULL if an unnamed mutex is being created. If this parameter is NULL a new mutex is always created.
HANDLE Return Value	Handle to new or existing mutex, or NULL on failure. GetLastError returns ERROR_ALREADY_EXISTS if an existing mutex was opened.

A thread owns a mutex from the time the thread's call to WaitForSingleObject returns until the thread calls ReleaseMutex. In other words, the thread owns the mutex while the mutex is signaled. A mutex can be initially created:

- **Signaled.** In this case, the thread that creates the mutex owns the mutex. All other threads calling WaitForSingleObject will block until the thread that owns the mutex calls ReleaseMutex.
- **Non-signaled.** The thread that creates the mutex does not own the mutex—in fact, no thread owns the mutex. The first thread that calls WaitForSingleObject will not block and will take ownership of the mutex.

A thread that owns a mutex can terminate before calling ReleaseMutex. In this case, the next thread to call WaitForSingleObject will take ownership of the mutex. However, WaitForSingleObject will return WAIT_ABANDONED rather than WAIT_OBJECT_0.

Table 6.4 *ReleaseMutex—Changes a mutex's state to non-signaled*

ReleaseMutex	
HANDLE hMutex	Handle of the mutex to change to non-signaled
BOOL Return Value	TRUE on success, otherwise FALSE

Listing 6.2 shows how to use a mutex to control access to a global variable. The code performs the same function as Listing 6.1 but uses a mutex instead of a critical section. The mutex is created by calling CreateMutex, and the second parameter ('TRUE') specifies that that mutex is owned by the thread that creates it. Any other thread that calls WaitForSingleObject will block until the thread that created the mutex calls ReleaseMutex. In Listing 6.2, each thread calls WaitForSingleObject before accessing the global variable g_fValueMutex. One of the thread functions fc1 or fc2 will unblock when the function Listing 6_2 calls ReleaseMutex. The other function will unblock when ReleaseMutex is called by the other unblocked thread function.

Listing 6.2 *Using a mutex*

```
float g_fValueMutex = 10.0;
DWORD WINAPI fc1(LPVOID);
DWORD WINAPI fc2(LPVOID);
HANDLE hMutex;

void Listing6_2()
{
  HANDLE hThread1, hThread2;
  DWORD dwThreadID;
```

```
  g_fValueMutex = 10.0;
  // Create mutex that's initially owned by this thread
  hMutex = CreateMutex(NULL, TRUE, NULL);
  hThread1 = CreateThread(NULL, 0,
      fc1, NULL, 0, &dwThreadID);
  hThread2 = CreateThread(NULL, 0,
      fc2, NULL, 0, &dwThreadID);
  // Release Mutex to allow both threads to
  // execute their code.
  ReleaseMutex(hMutex);
  // Wait until thread 1 and thread 2 completes
  WaitForSingleObject(hThread1, INFINITE);
  WaitForSingleObject(hThread2, INFINITE);
  // Close handle for the mutex and threads
  CloseHandle(hMutex);
  CloseHandle(hThread1);
  CloseHandle(hThread2);
  cout << _T("Finished:") << g_fValueMutex << endl;
}

DWORD WINAPI fc1(LPVOID)
{
  WaitForSingleObject(hMutex, INFINITE);
  g_fValueMutex = g_fValueMutex * g_fValueMutex;
  ReleaseMutex(hMutex);
  return 0;
}

DWORD WINAPI fc2(LPVOID)
{
  WaitForSingleObject(hMutex, INFINITE);
  g_fValueMutex = (float)3.0 + g_fValueMutex;
  ReleaseMutex(hMutex);
  return 0;
}
```

The function `CreateMutex` allows the mutex to be named—the last parameter is a string pointer to the mutex's name. If `CreateMutex` is called with the name of an existing mutex, the existing mutex will be opened rather than creating a new mutex. In this case, `CreateMutex` returns success, but `GetLastError` will return `ERROR_ALREADY_EXISTS`. Many processes can use a named mutex, so this allows mutual exclusion between processes. Windows CE does not support the Win32 function `OpenMutex`; however, all the functionality of `OpenMutex` is available through `CreateMutex`. Listing 4.25 in Chapter 4 ("Property Databases and the Registry") shows how to use a named mutex.

Using Event Objects

Event kernel objects are used to allow a thread to block until another thread has completed a task. For example, one thread may be reading data from the

Internet, and other threads can use an event to block until all the data has been read. Events can either be 'manual-reset' or 'auto-reset', and the type of event affects how threads blocking on the event behave.

- *Manual-Reset Events:* When the event becomes signaled through a thread calling `SetEvent`, all threads blocking on the event will be unblocked. The event remains signaled until any thread calls `ResetEvent` at which point the event becomes non-signaled.
- *Auto-Reset Events:* When the event becomes signaled through a thread calling `SetEvent`, only one thread blocking on the event will be unblocked, at which point the event will automatically become non-signaled.

Events are created through a call to `CreateEvent` (Table 6.5). This function allows both manual-reset and auto-reset events to be created with either a signaled or non-signaled state. Unlike mutex objects, events are not owned by a thread, so any thread can change the signaled state once it has a handle to the event. As with all kernel objects, `CloseHandle` should be called on the event handle when it is finished with.

Table 6.5 *CreateEvent—Creates a new event or opens an existing event*

CreateEvent	
`LPSECURITY_ATTRIBUTES lpMutexAttributes`	Not supported, pass as `NULL`.
`BOOL bManualReset`	`TRUE` to create a manual-reset event, or `FALSE` for an auto-reset event.
`BOOL bInitialState`	`TRUE` if the event is to be initially signaled, or `FALSE` if the event is to be initially non-signaled.
`LPTSTR lpName`	String containing name of event, or `NULL` if an unnamed event is being created. If this parameter is `NULL` a new event is always created.
`HANDLE Return Value`	Handle to new or existing event, or `NULL` on failure. `GetLastError` returns `ERROR_ALREADY_EXISTS` if an existing mutex was opened.

The only way to change an event's state to signaled is to call `SetEvent`. This function takes a single argument that is the handle to the event, and returns `TRUE` on success, or `FALSE` for failure. Threads don't need to explicitly change an event's state to non-signaled since this happens automatically when the first thread unblocks. Manual events can be set to non-signaled through calling the `ResetEvent` function. This function takes a single argument that is the handle to the event and returns a Boolean indicating success or failure.

A third function, `PulseEvent`, is used primarily with manual events. This function sets an event's state to signaled, and then immediately sets it to non-signaled. All threads that are blocked on the event are unblocked. However,

any threads that subsequently call `WaitForSingleObject` will block on the event until either `PulseEvent` or `SetEvent` are called.

As an example of when an event may be used, consider the code in Listing 6.3. The function `Listing6_3` declares a local variable structure '`threadInfo`', initializes the structure, and passes a pointer to a new thread through the `CreateThread` function. The thread function takes a copy of the structure pointed to by `lpThreadInfo` into a local structure variable called `tInfo`. Surprisingly, this thread function fails most of the time, since the thread function receives garbage in the structure that is passed to it. This is a classic synchronization problem—the function that creates the threads returns and the stack space occupied structure is reused when the thread goes on to call other functions. By the time the thread does execute, its pointer refers to a structure that is long gone (Figure 6.5).

Listing 6.3 *Thread creation that requires an event for synchronization*

```
typedef struct tagTHREADINFO {
    DWORD dwVal1, dwVal2;
} THREADINFO, *LPTHREADINFO;

DWORD WINAPI ThreadFunc(LPVOID lpThreadInfo);

void Listing6_3()
{
  THREADINFO threadInfo;
  HANDLE hThread;
  DWORD dwThreadId;

  threadInfo.dwVal1 = 20;
  threadInfo.dwVal2 = 40;

  hThread = CreateThread(NULL, 0, ThreadFunc,
        (LPVOID)&threadInfo, 0, &dwThreadId);
  CloseHandle(hThread);
}

DWORD ThreadFunc(LPVOID lpThreadInfo)
{
  LONG lResult;
  THREADINFO tInfo = *((LPTHREADINFO)lpThreadInfo);
  lResult = tInfo.dwVal1 * tInfo.dwVal2;
  cout << _T("Result: ") << lResult << endl;
  return 0;
}
```

This problem can be fixed by creating a non-signaled event in the function `Listing3_4`, and having the `Listing3_4` function block after creating the thread. The thread function `ThreadFunc` can then signal the event once it has taken a copy of the structure. This is shown in Listing 6.4, with the lines of

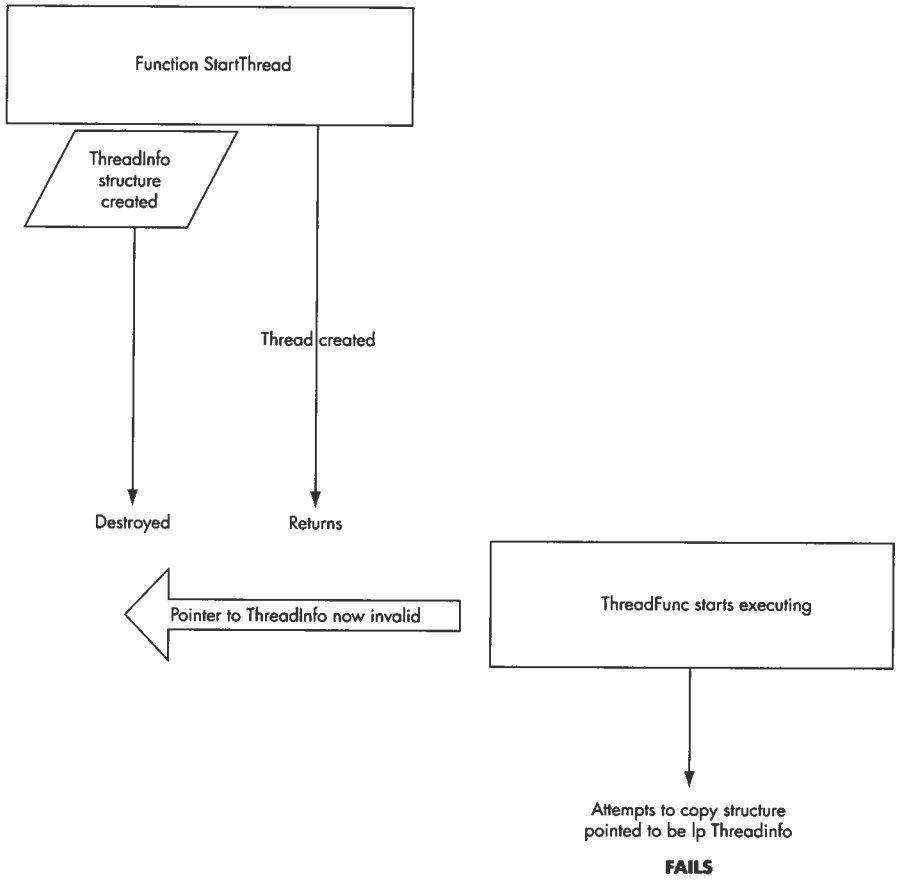


Figure 6.5 *Problem passing structure pointer to thread function*

code added for synchronization shown in bold. The event has its signal state changed just once with only a single thread waiting on it. Therefore, it does not matter if an auto-reset or manual-reset event is created. Figure 6.6 shows the program flow with this corrected code.

Listing 6.4 *Thread creation requiring an event for synchronization*

```
typedef struct tagTHREADINFO {
  DWORD dwVal1, dwVal2;
}THREADINFO, *LPTHREADINFO;

DWORD WINAPI ThreadFunc2(LPVOID lpThreadInfo);
HANDLE hEvent;
```

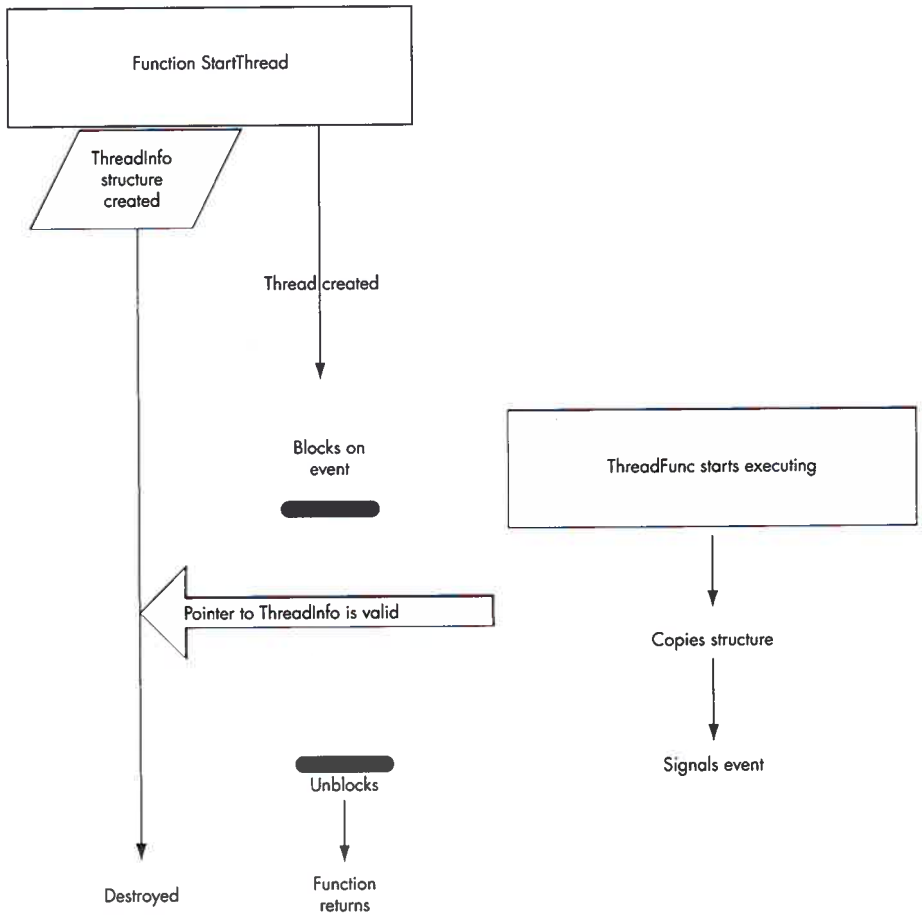


Figure 6.6 *Synchronized code for creating thread*

```
void Listing6_4()
{
  THREADINFO threadInfo;
  HANDLE hThread;
  DWORD dwThreadId;

  threadInfo.dwVal1 = 20;
  threadInfo.dwVal2 = 40;

  hEvent = CreateEvent(NULL,
      TRUE,    // manual event
      FALSE, // initially non-signaled
      NULL); // no name
```

```
  hThread = CreateThread(NULL, 0, ThreadFunc2,
        (LPVOID)&threadInfo, 0, &dwThreadId);
  WaitForSingleObject(hEvent, INFINITE);
  CloseHandle(hEvent);
  CloseHandle(hThread);
}

DWORD ThreadFunc2(LPVOID lpThreadInfo)
{
  LONG lResult;
  THREADINFO tInfo = *((LPTHREADINFO)lpThreadInfo);
  // Unblock Listing6_4 now the structure has been copied
  SetEvent(hEvent);
  lResult = tInfo.dwVal1 * tInfo.dwVal2;
  cout << _T("Result: ") << lResult << endl;
  return 0;
}
```

Events can be named through the last parameter to `CreateEvent`. This allows an event to synchronize threads in different processes. Windows CE does not support the `OpenEvent` function, but `CreateEvent` can be passed the name of an existing event and it will be opened. `GetLastError` will return `ERROR_ALREADY_EXISTS`.

Using Semaphores

Semaphores are an integer variable used to count in a synchronized way. Semaphores are often used to control access to a limited resource. For example, if your Windows CE device had two serial communications ports, you might use a semaphore to ensure that an application blocks if all communications ports are in use, and then un-blocks when one is freed. A semaphore object is signaled when less than the maximum number of resources is in use, and non-signaled when all the resources are in use. Semaphores were introduced in Windows CE 3.0. The following steps are used when using a semaphore:

- The semaphore is created or opened using the `CreateSemaphore` function (Table 6.6). This function is passed the maximum number of available resources and the initial number of resources in use.
- A thread calls `WaitForSingleObject` on the semaphore handle when it needs a resource. The resource count is automatically incremented when `WaitForSingleObject` returns.
- A thread calls `ReleaseSemaphore` when it has finished with the resource. The resource count is decremented.
- The function `CloseHandle` is called when the thread has finished with the semaphore.

Table 6.6 *CreateSemaphore—Creates a new semaphore or opens an existing semaphore*

CreateSemaphore	
`LPSECURITY_ATTRIBUTES lpMutexAttributes`	Not supported, pass as `NULL`.
`LONG lInitialCount`	Initial count (usually 0).
`LONG lMaximumCount`	Maximum count, for example, maximum number of available resources.
`LPTSTR lpName`	String containing name of semaphore, or `NULL` if an unnamed semaphore is being created. If this parameter is `NULL` a new semaphore is always created.
`HANDLE Return Value`	Handle to new or existing semaphore, or `NULL` on failure. `GetLastError` returns `ERROR_ALREADY_EXISTS` if an existing semaphore was opened.

A thread should call `WaitForSingleObject` to increment a semaphore's count, and this call will block if the semaphore has reached its maximum count. The function `ReleaseSemaphore` (Table 6.7) is used to decrement the semaphore's count. As a side effect, this function also returns the semaphore's count before the call to `ReleaseSemaphore` is made.

Table 6.7 *ReleaseSemaphore—Decrements a semaphore's count*

ReleaseSemaphore	
`HANDLE hSemaphore`	Semaphore's handle to decrement
`LONG lReleaseCount`	Pointer to a `LONG` that contains the *previous* count before `ReleaseSemaphore` decremented the count
`BOOL Return Value`	`TRUE` for success, otherwise `FALSE`

Interestingly, Windows does not have a function for determining the number of resources in use. The only way to determine this value is to call `WaitForSingleObject` with a zero timeout. If `WaitForSingleObject` returns `WAIT_TIMEOUT`, the semaphore is non-signaled, and the maximum number of resources is in use. For any other return value, `ReleaseSemaphore` is called, and the number of resources in use is returned in the `lReleaseCount` variable. Note that the release count is one greater than the actual number of resources in use—the call to `WaitForSingleObject` incremented the count.

Semaphores can be named through the last parameter to `CreateSemaphore`. This allows a semaphore count to be used by threads in different processes. Windows CE does not support the `OpenSemaphore` function, but `CreateSemaphore` can be passed the name of an existing semaphore and it will be opened. In this situation, `GetLastError` will return `ERROR_ALREADY_EXISTS`.

Selecting the Correct Synchronization Technique

The following describes the typical situations in which the various synchronization techniques are used.

- Mutex objects are used to stop two threads from attempting to access some shared resource at the same time. Critical sections can be used within a process.
- Event objects are used to allow one or more threads to block until another thread has completed a task.
- Semaphore objects are used when synchronized counting is required with some given maximum value.

Thread Local Storage and Dynamic Link Libraries

When writing a multithreaded application you not only need to ensure that your code is correctly synchronized, but also that any libraries you call are also written to be multithreaded. If your code calls functions in a Dynamic Link Library that are designed only for single-threaded calls, you can run into deadlock, race conditions, and other synchronization problems.

If you know that a library is single-threaded, you need to ensure that you always call functions in that library on the same thread. That way, two threads will not be actively calling functions in the library at the same time—this is called *serialization*.

When writing a library yourself you need to decide whether you want it to be single- or multithreaded. A library written to be multithreaded must use the synchronization techniques described in previous sections—this makes it *thread-safe*. If you are writing a *thread-safe* library, you may have to use Thread Local Storage (TLS) to manage global variables. Consider the situation of the `errno` variable in the C run-time library. This variable contains the last error number encountered when calling a C run-time function such as `fopen`. In a multithreaded application, two threads may call `fopen` at much the same time, and if there is only a single `errno` variable, one thread could end up using the return result from the other thread's `fopen` call (Figure 6.7). This can happen if a multithreaded application calls the single-threaded C run-time library. The solution is for each thread to have its own copy of `errno`. This means that when the thread is created, the `errno` variable must be created, and when the thread terminates, the `errno` variable must be destroyed. Windows CE provides Thread Local Storage (TLS) to solve this problem. TLS is most often used in Dynamic Link Libraries, but it can also be used in EXEs.

When an application starts up, Windows CE creates 64 slots into which a `DWORD` value can be stored for each thread in that application. A DLL can reserve one of these slots for its own use by calling `TlsAlloc`. This function then

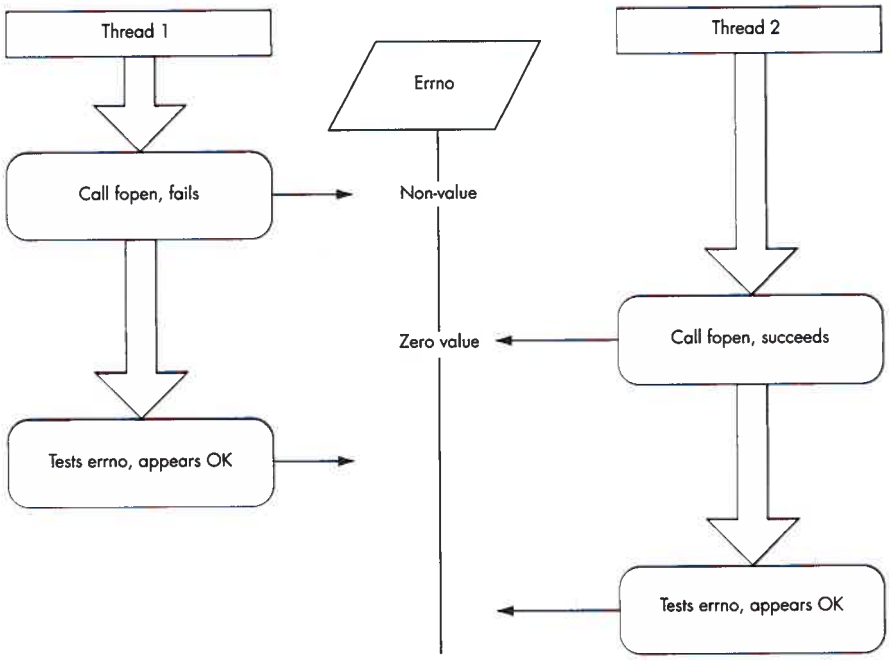


Figure 6.7 *Single-threaded C run-time calls from a multithreaded application*

returns the next available slot number. The function `TlsSetValue` can be used to store a `DWORD` value into a slot for the thread on which `TlsSetValue` is called, and the function is passed the slot number that was returned from `TlsAlloc`. Note that separate `DWORD` values can be stored for each thread in an application using this technique. At some later stage, the thread can call `TlsGetValue` to retrieve the value held in the given slot for that thread. Finally, when the slot is finished with, `TlsFree` is called and is passed the slot number returned from calling `TlsAlloc`.

Slots are in quite short supply, so a DLL will typically only request a single slot by calling `TlsAlloc`. If the thread needs to store more than a single `DWORD`, it will generally use dynamic memory allocation (see Chapter 12), and store a pointer to this memory using `TlsSetValue`. DLLs receive notification of when a thread is created in the application the DLL is mapped into—Windows CE calls the DLL's `DllMain` function, passing the `DLL_THREAD_ATTACH` value in the `fdwReason` parameter. Likewise, a DLL is notified when a thread terminates with the `fdwReason` parameter set to `DLL_THREAD_DETACH`.

The code in Listing 6.5 shows how to use TLS for a DLL that needs to maintain a string buffer for each thread calling into the library. The code shows an implementation of `DllMain`, and this function is called when

- fdwReason = DLL_PROCESS_ATTACH: The DLL is first loaded by the process.
- fdwReason = DLL_THREAD_ATTACH: A thread is created by any code in the process, not just threads created by the DLL.
- fdwReason = DLL_THREAD_DETACH: A thread in the process terminates.
- fdwReason = DLL_PROCESS_DETACH: The process terminates.

A global variable g_dwTlsIndex is declared, and this will contain the slot number for this DLL obtained from calling TlsAlloc when the DLL is first loaded. The function TlsAlloc returns the value TLS_OUT_OF_INDEXES if no more slots are available. Returning FALSE from DllMain when the reason code is DLL_PROCESS_ATTACH causes the loading of the DLL to fail.

Listing 6.5 *Using TLS data in a Dynamic Link Library*

```
DWORD g_dwTlsIndex = TLS_OUT_OF_INDEXES;

BOOL WINAPI DllMain(HINSTANCE hinstDLL,
        DWORD fdwReason, LPVOID fImpLoad)
{
  LPTSTR lpszStr;

  switch(fdwReason) {
  case DLL_PROCESS_ATTACH:
    g_dwTlsIndex = TlsAlloc();
    if(g_dwTlsIndex == TLS_OUT_OF_INDEXES)
      return (FALSE);
    break;
  case DLL_THREAD_ATTACH:
      break;
  case DLL_THREAD_DETACH:
    if(g_dwTlsIndex != TLS_OUT_OF_INDEXES)
    {
      lpszStr =
        (LPTSTR)TlsGetValue(g_dwTlsIndex);
      if(lpszStr != NULL)
        HeapFree(GetProcessHeap(),
          0, lpszStr);
    }
    break;
  case DLL_PROCESS_DETACH:
    if(g_dwTlsIndex != TLS_OUT_OF_INDEXES)
    {
      lpszStr =
        (LPTSTR)TlsGetValue(g_dwTlsIndex);
      if(lpszStr != NULL)
        HeapFree(GetProcessHeap(),
          0, lpszStr);
```

```
      TlsFree(g_dwTlsIndex);
    }
    break;
  }
  return TRUE;
}
```

The code in Listing 6.5 does not allocate memory when the reason code is DLL_THREAD_ATTACH. It is more efficient to only allocate the memory associated with TLS the first time it is needed. The code in Listing 6.6 shows how a function can determine if memory for this thread has already been allocated. The TlsGetValue function is called and is passed the slot number returned from TlsAlloc. If this value is NULL, the memory has not yet been allocated, so HeapAlloc is called to allocate a buffer of 40 bytes in the default heap. The pointer to this newly allocated memory is stored as TLS using the TlsSetValue function. Remember that this allocation will occur for each thread that calls FunctionX.

Listing 6.6 *Allocating TLS data for a thread*

```
LPTSTR FunctionX()
{
  LPTSTR lpszStr = (LPTSTR)TlsGetValue(g_dwTlsIndex);
  if(lpszStr == NULL)
  {
    lpszStr = (LPTSTR)HeapAlloc(GetProcessHeap(),
        0, 40);
    TlsSetValue(g_dwTlsIndex, lpszStr);
  }
  // Now use lpszStr in some way...
}
```

The memory allocated and stored in TLS must be freed when the thread terminates. This is done when DllMain is called with the reason code DLL_THREAD_DETACH. Since DllMain is called using the thread that is being terminated, calling TlsGetValue will return the DWORD associated with the terminating thread. This code (from Listing 6.5) gets data associated with the slot and thread, and if this is non-null, frees the data.

```
case DLL_THREAD_DETACH:
  if(g_dwTlsIndex != TLS_OUT_OF_INDEXES)
  {
    lpszStr =
      (LPTSTR)TlsGetValue(g_dwTlsIndex);
    if(lpszStr != NULL)
      HeapFree(GetProcessHeap(),
        0, lpszStr);
  }
  break;
```

When the process unloads the DLL, the slot number has to be freed. In Listing 6.5 a check is also made to see if the data associated with the thread being used to unload the library has been freed, then a call is made to `TlsFree` to free the slot.

```
case DLL_PROCESS_DETACH:
  if(g_dwTlsIndex != TLS_OUT_OF_INDEXES)
  {
    lpszStr =
      (LPTSTR)TlsGetValue(g_dwTlsIndex);
    if(lpszStr != NULL)
      HeapFree(GetProcessHeap(),
        0, lpszStr);
    TlsFree(g_dwTlsIndex);
  }
  break;
```

The technique described above for TLS is called 'dynamic TLS', since memory is allocated and de-allocated dynamically for each thread. In Windows NT/98/2000, 'static TLS' is also supported through the `#pragma data_seg` compiler directive. Any variable declarations placed between `#pragma data_seg` compiler directives will be duplicated for each thread. Static TLS is not supported in Windows CE.

As an aside, Windows CE 3.0 now supports the `DisableThreadLibraryCalls` function. Calling this function disables `DllMain` being called with the reason codes `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH`. The function takes a single argument which is the DLL's module or instance handle. This can reduce code size and, for processes that create large number of threads, improve performance. Of course, you don't want to call `DisableThreadLibraryCalls` for a DLL that implements TLS using the techniques described here. The best place to `DisableThreadLibraryCalls` is in `DLLMain` when the reason code is `DLL_PROCESS_ATTACH`.

```
case DLL_PROCESS_ATTACH:
  DisableThreadLibraryCalls(hinstDLL);
  break;
```

Conclusion

This chapter has shown various illustrations of why thread synchronization is so important, and described how critical sections, mutex, event, and semaphore objects can be used for synchronization. Any multithreaded application will need to employ such techniques. Further, single- or multithreaded applications that need to synchronize with other applications require synchronization. Finally, if you are developing multithreaded DLLs, you may need to use Thread Local Storage (TLS) for global or dynamic data.

SEVEN

Notifications

The notification functions described in this chapter can be used to run an application at a particular time, or in response to an event such as completion of synchronization or a serial connection being made. Alternatively, the functions can be used to notify the user of such events through, for example, a flashing LED. The notification functions will operate even if the Windows CE device is suspended. The events for which a notification can be given include:

- When data synchronization finishes
- When a PC Card device is changed
- When an RS232 connection is made
- When the system time is changed
- When a full device data restore completes

Users can be notified in a variety of ways, and should be allowed to specify their preference. Notification can occur by:

- Flashing the LED
- Vibrating the device
- Displaying the user notification dialog box
- Playing a sound

Windows CE operating system versions prior to 2.12 should use the notification functions `CeRunAppAtTime`, `CeRunAppAtEvent`, and `CeSetUserNotification`. These are described in the first part of this chapter. In Windows CE 2.12, 3.0, and later, these functions are replaced with the single function `CeSetUserNotificationEx`. This function can be used when you don't need compatibility with earlier operating system versions, or when you need the additional functionality it provides. It is described in the section "`CeSetUserNotificationEx`".

Running an Application at a Specified Time

The function CeRunAppAtTime sets an application to be run at a time specified in a SYSTEMTIME structure. Listing 7.1 shows how this function can be called to run Pocket Word at 7.20AM on the current day. Note that the file notify.h must be included when using notification functions. The code gets the current local time through calling GetLocalTime, and sets the hour to 7 and minute to 20. The call to CeRunAppAtTime is passed the name of the application to run and the time to run it.

Listing 7.1 *Runs an application at a specified time*

```
#include <notify.h>

void Listing7_1()
{
  SYSTEMTIME sysTime;

  GetLocalTime(&sysTime);
  sysTime.wHour = 7;
  sysTime.wMinute = 20;
  if(!CeRunAppAtTime(
      _T("\\Windows\\Pword.exe"), &sysTime))
    cout  << _T("Cannot set application to run")
          << endl;
  else
    cout  << _T("App set to run at specified time")
          << endl;
}
```

Windows CE will run the application at the specified time and pass the command line parameter 'AppRunAtTime'. For this reason, Pocket Word will prompt you to create a new file called 'AppRunAtTime.Pwd' when it runs. If the time specified in the SYSTEMTIME structure is in the past the application will run immediately.

An application can only have a single CeRunAppAtTime request associated with it. If another call is made to CeRunAppAtTime for the same application, the previous request is overwritten with the new time. A CeRunAppAtTime request for an application can be removed by passing NULL pointer for the SYSTEMTIME pointer.

Using Mini-Applications with Notification

In general it is best not to run large applications using CeRunAppAtTime, since the user may be confused by a new application suddenly appearing and may

cause an out-of-memory error. Instead, you should create a 'mini-application' with no user interface and have this application run at the specified time. The 'mini-application' can then notify the main application through a private message of the event, or perhaps perform some scheduled task.

Listing 7.2 shows the code for a 'mini-application' called `Notify.exe`. This is a Windows CE application with a `WinMain` function that registers a new windows message using the `RegisterWindowMessage` function. The `RegisterWindowMessage` function is passed a string and returns a message number. Any application that calls `RegisterWindowMessage` with the same string will always receive back the same message number, and so the message can be used for communication between applications. Next, the `WinMain` function calls `SendMessage` using the special window handle `HWND_BROADCAST`. This sends the message number in `nNotifyMsg` to all top-level windows. `WinMain` returns and the mini-application ends. The code for `Notify.exe` is located on the CDROM in the folder `\Notify`.

Listing 7.2 *Notify.exe—'Mini-application' used for notification*

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPWSTR lpCmdLine, int nShowCmd)
{
  UINT nNotifyMsg =
    RegisterWindowMessage(_T("MSG_NOTIFY"));

  SendMessage(HWND_BROADCAST, nNotifyMsg, 0, 0);
  return 0;
}
```

An application can use the `CeRunAppAtTime` function to run `Notify.exe` at a specified time:

```
CeRunAppAtTime(_T("\\Notify.exe"), &sysTime))
```

To respond to the broadcast `SendMessage` from `Notify.exe` an application must use `RegisterWindowMessage` when it first starts, using the same string as used in `Notify.exe`.

```
nNotifyMsg = RegisterWindowMessage(_T("MSG_NOTIFY"));
```

Next, the application must add code to the window message procedure for its top-level, main application window to handle the message number held in `nNotifyMsg`.

```
if(message == nNotifyMsg)
{
  cout << "Notification: Application has run"
       << endl;
}
```

The technique described here only responds to Notify.exe if the application in question is running, otherwise the broadcast message will be ignored. The mini-application can check if the application is running, and if not, call CreateProcess to run it.

```
HWND hWnd = FindWindow(_T("Examples"), NULL);
PROCESS_INFORMATION pi;
if(hWnd == NULL)
  CreateProcess(_T("\\Examples.exe"), NULL,
    NULL, NULL, FALSE,0, NULL,
    NULL, NULL, &pi);
SendMessage(HWND_BROADCAST, nNotifyMsg, 0, 0);
```

The function FindWindow is passed the class name 'Examples' of the main application window in 'Examples.exe'. A returned NULL handle indicates that the window could not be found, and therefore the application is not running. In this case, CreateProcess is called to run Examples.exe. (See Chapter 5 for more information on CreateProcess.) Unfortunately, the application Examples.exe will not receive the notification message! This is because CreateProcess returns *before* the application has initialized and created the main window. This is a classic synchronization problem. A simple solution would be to add a 'while' loop after CreateProcess but before SendMessage.

```
if(hWnd == NULL)
{
  CreateProcess(_T("\\Examples.exe"), NULL,
    NULL, NULL, FALSE,0, NULL,
    NULL, NULL, &pi);
  while(FindWindow(_T("Examples"), NULL) == NULL)
    Sleep(100);
}
SendMessage(HWND_BROADCAST, nNotifyMsg, 0, 0);
```

In this case, the program loops until FindWindow returns a non-NULL handle, and sleeps the thread for 100 milliseconds on each loop iteration to avoid hogging the CPU. However, this solution is not ideal because:

- The loop will continue forever if the main application window in Examples.exe could not be created.
- The call to the Sleep function introduces unnecessary delays.
- Although the window will have been created when SendMessage is called, the WM_CREATE message may not have been processed. Therefore, the window may not be properly initialized when the notification is received.

The correct solution is to use a synchronization event, which is shown in Listing 7.3. An event is created which is manually signaled (the FALSE parameter) that will be initially non-signaled (the 0 parameter). The event is given a name so that the same event can be used in the Example.exe application.

The WaitForSingleObject function is used to wait on the event to be signaled, with a timeout of 5000 milliseconds. The Example application will signal this event when initialization is complete and the application is ready to receive a notification. Events are described in more detail in Chapter 6.

Listing 7.3 *Notify.exe with synchronization*

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPWSTR lpCmdLine, int nShowCmd)
{
  UINT nNotifyMsg =
    RegisterWindowMessage(_T("MSG_NOTIFY"));

  HWND hWnd = FindWindow(_T("Examples"), NULL);
  PROCESS_INFORMATION pi;
  HANDLE hEvent;

  if(hWnd == NULL)
  {
    // create non-signaled event
    hEvent = CreateEvent(NULL, TRUE,
        0, _T("Examples_Event"));
    if(CreateProcess(_T("\\Examples.exe"),
        NULL, NULL, NULL,
        FALSE,0, NULL, NULL, NULL, &pi))
    {
      CloseHandle(pi.hProcess);
      CloseHandle(pi.hThread);
      if(WaitForSingleObject(hEvent, 5000)
          == WAIT_FAILED)
        MessageBox(NULL,
          _T("Example start failed "),
          NULL, MB_OK);
      CloseHandle(hEvent);
    }
    else
      MessageBox(NULL,
        _T("Could not start Example.exe"),
        NULL, MB_OK);

  }
  SendMessage(HWND_BROADCAST, nNotifyMsg, 0, 0);
  return 0;

}
```

The Example.exe application will signal the event when initialization is completed, which could be, for example, when WM_CREATE has been handled.

```
case WM_CREATE:
  HANDLE hEvent;
  hEvent = CreateEvent(NULL,
    TRUE, 0, _T("Examples_Event"));
  ResetEvent(hEvent);
  CloseHandle(hEvent);
  break;
```

The `CreateEvent` function is called, which will open the event created in `Notify.exe` as it is passed the same name. The `ResetEvent` function is called to signal the event, and this will unblock `Notify.exe`'s call to `WaitForSingleObject`.

An alternative approach would be for `Example.exe` to run itself from `CeRunAppAtTime`. Then, when the `Example.exe` is run at the specified time, it would need to determine if it is the first instance running by calling `FindWindow`. If it were the first instance, `Example.exe` would go ahead and do the necessary processing and then quit. If it is the second instance, it should notify the first instance using `SendMessage`, and then quit. The first instance can then perform the necessary processing. This approach has several disadvantages:

- Loading the application may be slow if it is large.
- The application may take significant amounts of memory, and for a brief time, there may be two instances requiring up to twice as much memory.
- The application will need to work out whether the user interface should be displayed depending on how the application was started.

Starting an Application on an Event

The function `CeRunAppAtEvent` allows a program to be run when one of the following events occur:

- `NOTIFICATION_EVENT_SYNC_END`. When data synchronization finishes.
- `NOTIFICATION_EVENT_DEVICE_CHANGE`. When a PC Card device is changed.
- `NOTIFICATION_EVENT_RS232_DETECTED`. When an RS232 connection is made.
- `NOTIFICATION_EVENT_TIME_CHANGE`. When the system time is changed.
- `NOTIFICATION_EVENT_RESTORE_END`. When a full device restore completes.

Listing 7.4 shows a call to `CeRunAppAtEvent` that sets `Notify.exe` to run when `ActiveSync` synchronization completes.

Listing 7.4 *Runs application on an event*

```
void Listing7_4()
{
  if(!CeRunAppAtEvent(_T("\\Notify.exe"),
            NOTIFICATION_EVENT_SYNC_END))
    cout  << _T("Cannot set application to run")
          << endl;
  else
    cout
      << _T("Notify.exe will run when sync finishes")
      << endl;
}
```

The application will be run with a command line string whose value depends on the event being used, and these strings as shown in Table 7.1.

Table 7.1 *Command line strings used with CeRunAppAtEvent*

Constant	Value
APP_RUN_AFTER_SYNC	"AppRunAfterSync"
APP_RUN_AT_DEVICE_CHANGE	"AppRunDeviceChange"
APP_RUN_AT_RS232_DETECT	"AppRunAtRs232Detect"
APP_RUN_AFTER_RESTORE	"AppRunAfterRestore"

The application specified in `CeRunAppAtEvent` will be run each time the specified event occurs. All events associated with an application can be removed by calling the function `CeRunAppAtEvent` with `NOTIFICATION_EVENT_NONE` as the last parameter (Listing 7.5).

Listing 7.5 *Removes an application event*

```
void Listing7_5()
{
  if(!CeRunAppAtEvent(_T("\\Notify.exe"),
           NOTIFICATION_EVENT_NONE))
    cout << _T("Cannot stop application event.")
         << endl;
  else
    cout << _T("Application event removed.") << endl;
}
```

Manually Controlling the LED

The notification functions use the LED as one way to notify the user of an event, but sometimes it is necessary to control the LED yourself. For example, you might want to notify the user of an event not supported by the notification functions. The `NLedGetDeviceInfo` function is used to determine the number of LEDs on the device—It is conceivable that a special device may have more than one LED, and some devices may not have any at all. The `NLedSetDevice` function is used to turn the LED on and off. Both these functions interact with the LED driver written by the device's manufacturer.

There are various options that an LED driver can support beyond the simple default blinking behavior. The `NLedSetDevice` function allows the following options to be set:

- Total Cycle Time: The total time the LED will blink before turning itself off
- The time for which the LED will be on
- The time for which the LED will be off
- The on meta-cycle time
- The off meta-cycle time

A LED can simply blink on and off using the on and off times, or it can perform a more complex sequence using the meta-cycle times. With a meta-cycle, the LED will blink for the meta-cycle time, and then turn off completely for the meta-cycle off time. It will then blink the LED for the on meta-cycle time, and so on for the total cycle time. Before you start implementing Morse code for the LED, you should note that most devices only support simple on-off blinking.

The functions `NLedGetDeviceInfo` and `NLedSetDevice` are implemented in `coredll.dll`, but are not generally declared in SDK header files. Therefore, you will need to add function prototypes. Also, the functions use structures that are declared in the header file `NLed.H`:

```
#include <NLed.h>
extern "C"
{
  BOOL NLedGetDeviceInfo(INT nId, PVOID pOutput);
  BOOL NLedSetDevice(INT nId, PVOID pOutput);
}
```

First, you will need to determine the number of LEDs present on the device, and then get the capabilities of the LED, using the function `NLedGetDeviceInfo`. The function takes an identifier as the first argument that specifies what information is being requested, and a pointer to an appropriate structure to receive the information in the second parameter. Table 7.2 shows the identifiers and the corresponding structures.

Table 7.2 *Identifiers and structures for NLedGetDeviceInfo*

Constant	Structure	Purpose
NLED_COUNT_INFO_ID	NLED_COUNT_INFO	Return the number of LEDs
NLED_SUPPORTS_INFO_ID	NLED_SUPPORTS_INFO	Determine LED capabilities
NLED_SETTINGS_INFO_ID	NLED_SETTINGS_INFO	Return current LED settings

Listing 7.6 shows calling NLedGetDeviceInfo first to determine the number of LEDs, which is returned in the cLeds member of NLED_COUNT_INFO. Assuming there is one, the next call to NLedGetDeviceInfo will get the characteristics associated with LED number zero (the first). To do this, the NLED_SUPPORTS_INFO structure member LedNum is initialized with the LED number, and then the call is made.

Listing 7.6 *Determines LED capabilites*

```
void Listing7_6()
{
  NLED_COUNT_INFO nci;
  NLED_SUPPORTS_INFO  nsup;

  if(!NLedGetDeviceInfo(NLED_COUNT_INFO_ID,
                    (PVOID) &nci))
  {
    cout  << _T("Could not get LED information")
          << endl;
    return;
  }
  cout  << _T("Number of LEDs: ") << (int)nci.cLeds
        << endl;
  memset(&nsup, 0, sizeof(nsup));
  nsup.LedNum = 0; // get information on first LED
  if(!NLedGetDeviceInfo(NLED_SUPPORTS_INFO_ID,
    (PVOID) &nsup))
  {
    cout  << _T("Could not get LED support options")
          << endl;
    return;
  }
  cout  << _T("Cycle Adjust:") //0 = off 1 = on 2 = blink
        << nsup.lCycleAdjust << endl;
  cout << _T("Adj. Total Cycle Time:")
        << nsup.fAdjustTotalCycleTime << endl;
  cout << _T("Separate On Time:")
        << nsup.fAdjustOnTime << endl;
  cout << _T("Separate Off Time:"
        << nsup.fAdjustOffTime << endl;
```

```
  cout << _T("Can Meta Cycle On:"
        << nsup.fMetaCycleOn << endl;
  cout << _T("Can Meta Cycle Off:")
        << nsup.fMetaCycleOff << endl;
}
```

The `lCycleAdjust` member indicates whether the LED can be turned on and off, or be made to blink. The remaining members are BOOL values indicating which timings, if any, can be changed.

The code in Listing 7.7 is used to toggle the LED between blinking and not blinking. The `NLED_SETTINGS_INFO` structure member `LedNum` is initialized with the LED number set, and `OffOnBlink` will be set to 2 to start blinking or 0 to stop blinking. This structure has other members to change cycle times and so on, but they are not used in this example.

Listing 7.7 *Toggles LED blinking status*

```
void Listing7_7()
{
  NLED_SETTINGS_INFO nsi;
  static int nLastSetting = 0;          // initially off

  if(nLastSetting == 0)
    nLastSetting = 2;  // blink
  else
    nLastSetting = 0;  // off
    nsi.LedNum = 0;
  nsi.OffOnBlink = nLastSetting;
  if(!NLedSetDevice(NLED_SETTINGS_INFO_ID, &nsi))
    cout << _T("Could not set LED settings") << endl;
}
```

User Notification

You can use the function `CeSetUserNotification` to notify at a given time using a flashing LED, dialog box, or other technique supported by the Windows CE device. This function will place an icon (the 'annunciator icon') in the tool box at the bottom left of the screen. When this icon is double-clicked by the user, an application specified in `CeSetUserNotification` will be run. This annunciator icon should be removed by calling `CeHandleAppNotifications`—it cannot be removed by the user.

The code in Listing 7.8 used `CeSetUserNotification` to notify the user at 7:15 on the current day by playing the WAV file `Alarm2.wav` repeatedly. The function returns a handle that can be used to further manipulate the notification. Table 7.3 describes the `CeSetUserNotification` parameters.

Listing 7.8 *Setting user notification*

```
void Listing7_8()
{
  HANDLE hNotify;
  SYSTEMTIME sysTime;
  CE_USER_NOTIFICATION ceNot;

  GetLocalTime(&sysTime);
  sysTime.wHour = 7;
  sysTime.wMinute= 15;

  ceNot.ActionFlags = PUN_SOUND | PUN_REPEAT;
  ceNot.pwszSound = _T("\\Windows\\Alarm2.wav");
  hNotify = CeSetUserNotification(
          NULL,
          _T("\\Notify.exe"),
          &sysTime,
          &ceNot);
  if(hNotify == NULL)
    cout  << _T("Could not set user notification")
          << endl;
  else
    cout << _T("User notification set") << endl;
}
```

The application specified in `pwszAppName` will be run when the annunciator icon is clicked by the user. The application (`Notify.exe` in Listing 7.8) will be passed the command line string `APP_RUN_TO_HANDLE_NOTIFICATION` and the notification handle (converted to a string).

Table 7.3 *CeSetUserNotification*

CeSetUserNotification	
`HANDLE hNotification`	Handle of the notification to modify, or `NULL` for a new notification.
`TCHAR *pwszAppName`	Name of the associated application. This does not have to be the application setting the notification.
`SYSTEMTIME *lpTime`	`SYSTEMTIME` structure specifying the time for the notification to occur.
`PCE_USER_NOTIFICATION lpUserNotification`	`CE_USER_NOTIFICATION` structure containing information on how to notify the user.
`HANDLE Return Value`	Returns a `HANDLE` to the event.

The `CE_USER_NOTIFICATION` structure specifies how the user will be notified by setting the `ActionFlags` member with one or more of the following flags shown in Table 7.4.

Table 7.4 *CE_USER_NOTIFICATION ActionFlags values*

Value	Description
PUN_LED	Flash the LED.
PUN_VIBRATE	Vibrate the device.
PUN_DIALOG	Display a dialog to the user. The CE_USER_NOTIFICATION members pwszDialogTitle and pwszDialogText specify the dialog's caption text and body text.
PUN_SOUND	Plays a WAV file specified in the CE_USER_NOTIFICATION member pwszSound.
PUN_REPEAT	Repeats playing the WAV file for around 10 to 15 seconds.

The application associated with the notification will be run when the user clicks the annunciator icon, and this application should remove the icon. This is done by calling the CeHandleAppNotifications function, passing in the name of the application associated with the notification (Listing 7.9).

Listing 7.9 *Removes the annunciator icon*

```
void Listing7_9()
{
  if(CeHandleAppNotifications(_T("\\Notify.exe")))
    cout << _T("Annunciator cleared") << endl;
  else
    cout << _T("Annunciator could not be cleared")
         << endl;
}
```

The handle returned from CeSetUserNotification can be used to modify or remove the notification as long as the notification time has not passed. A notification can be modified by passing the notification handle as the first argument, and passing in new values for the time or CE_USER_NOTIFICATION structure. A notification can be removed entirely by passing the handle to the CeClearUserNotification function.

```
  if(CeClearUserNotification (hNotify)))
    cout << _T("Notification cleared") << endl;
  else
    cout << _T("Notification could not be cleared")
         << endl;
```

Users can specify their preference on how they wish to be notified, and these preferences should be honored by your application. The function CeGet-UserNotificationPreferences can be used to display a dialog prompting the user for his or her preferred notification options. The dialog will then populate a CE_USER_NOTIFICATION structure with these preferences, and this structure can be passed to CeSetUserNotification to set the notification.

Note that the CE_USER_NOTIFICATION structure can be initialized *before* calling CeGetUserNotificationPreferences to set default values in the dialog box.

Listing 7.10 *Getting user preferences for notifications*

```
void Listing7_10(HWND hWnd)
{
  CE_USER_NOTIFICATION ceNot;
  TCHAR szSound[MAX_PATH + 1];

  ceNot.ActionFlags = PUN_SOUND | PUN_REPEAT;
  ceNot.pwszSound = szSound;
  ceNot.nMaxSound = MAX_PATH;

  if(!CeGetUserNotificationPreferences(hWnd, &ceNot))
    cout << _T("Could not get settings") << endl;
  else
  {
    if(ceNot.ActionFlags & PUN_SOUND)
    {
      cout << _T("SOUND:") << endl;
      if(ceNot.ActionFlags & PUN_REPEAT)
        cout << _T("Repeat") << endl;
      else
        cout << _T("Don't repeat") << endl;
      cout << _T("Sound: ") << ceNot.pwszSound
        << endl;
    }
    if(ceNot.ActionFlags & PUN_LED)
      cout << _T("FLASH") << endl;
    if(ceNot.ActionFlags & PUN_VIBRATE)
      cout << _T("VIBRATE") << endl;
    if(ceNot.ActionFlags & PUN_DIALOG)
      cout << _T("DIALOG") << endl;
  }
}
```

CeSetUserNotificationEx

So far, all the functions described in this chapter are available in the Windows CE operating system versions 2.0 and later. However, in Windows CE 2.12 and later, many of the notification functions (such as CeSetUserNotification, CeRunAppAtTime, and CeRunAppAtEvent) have been replaced with the single function CeSetUserNotificationEx. You should use this function if you do not require backwards compatibility with earlier Windows CE versions. CeSetUserNotificationEx provides additional capabilities not present in earlier operations systems, such as:

- Specifying a time period (start time and end time) during which a notification is active. With `CeSetUserNotification` a notification is active from the start time until it is removed.
- Specifying the command line arguments passed to an application launched by a notification rather than the standard arguments listed in Table 7.1.

Table 7.5 lists the `CeSetUserNotificationEx` arguments and return type. The function can be used to modify an existing notification by passing a valid notification handle as the first argument, or 0 to create a new notification.

Table 7.5 *CeSetUserNotificationEx notification function*

CeSetUserNotification	
`HANDLE hNotification,`	Handle of the notification to modify, or 0 for a new notification.
`CE_NOTIFICATION_TRIGGER *pcnt`	Structure defining the type of notification.
`CE_USER_NOTIFICATION *pceun`	Pointer to a user notification structure. This is the same structure used with `CeSetUserNotification`.
`HANDLE Return Value`	Returns a `HANDLE` to the event.

The `CE_NOTIFICATION_TRIGGER` structure defines what type of notification is being created and what the notification will do. Table 7.6 lists the structure members.

Table 7.6 *CE_NOTIFICATION_TRIGGER structure*

Member	**Purpose**
`DWORD dwSize`	Size of the structure in bytes.
`DWORD dwType`	Type of notification: `CNT_EVENT`—System event notification. `CNT_TIME`—Time-based notification. `CNT_PERIOD`—Period-based notification using `stStartTime` and `stEndTime`. `CNT_CLASSICTIME`—Same behavior as calling the `CeSetUserNotification` with standard command line values.
`DWORD dwEvent`	If `dwType == CNT_EVENT` this member is initialized with a standard event constant, see Table 7.1.
`WCHAR *lpszApplication`	Name of application to run.
`WCHAR *lpszArguments`	Arguments to be passed to application. Must be `NULL` if `dwType == CNT_CLASSICTIME`.
`SYSTEMTIME stStartTime`	Specifies the start time of the notification period.
`SYSTEMTIME stEndTime`	Specifies the end time of the notification period.

The code in Listing 7.11 shows how CeSetUserNotification can be called to run Pocket Word at a specified time (10.15PM on the current day) with no command line argument being passed. No user notifications are required, so the CE_USER_NOTIFICATION structure pointer is passed as NULL.

Listing 7.11 *Runs an application at a specified time using CeSetUserNotification*

```
void Listing7_11()
{
  CE_NOTIFICATION_TRIGGER unt;
  CE_USER_NOTIFICATION cen;

  SYSTEMTIME sysTime;

  GetLocalTime(&sysTime);
  sysTime.wHour = 22;
  sysTime.wMinute = 15;

  memset(&unt, 0, sizeof(unt));
  unt.dwSize = sizeof(unt);
  unt.dwType = CNT_TIME;
  unt.lpszApplication = _T("\\windows\\pword.exe");
  unt.lpszArguments = NULL; // no command line argument
  unt.stStartTime = sysTime;
  unt.stEndTime = sysTime;

  HANDLE hNotify = CeSetUserNotificationEx(0,
        &unt, NULL);
  if(hNotify == NULL)
    cout << _T("Could not set notification") << endl;
  else
    cout << _T("Notification set") << endl;
}
```

Conclusion

This chapter has shown how the notification functions can be used to inform the user or to run an application when a standard event occurs. Standard events include ActiveSync completing or an RS232 serial connection being made. The notify functions also allow an application to be run in response to a standard event or at a given time.

```
    else
      cout << _T("Checksum OK");
}
```

The function ParseRMC in Listing 9.4b calculates a checksum for the sentence. The checksum is obtained by starting with a zero integer value in dwCheckSum and then performing an exclusive or (XOR) operation on dwCheckSum and each character in the sentence. All characters between the '$' marking the start of the sentence and '*' marking the start of the checksum value are used. Note that the '$' and '*' characters themselves are not included.

```
for(UINT i = 1; i < wcslen(szSentence) &&
      szSentence[i] != '*'; i++)
  dwCheckSum ^= szSentence[i];
```

Once the data items have been displayed, the reported checksum value is extracted from the sentence, and this is compared to the calculated checksum value. The checksum value in the sentence is sent as a two-digit hexadecimal value. The function wcstoul is used to convert the string representation of the checksum value into a binary value that is stored in the variable dwSentenceCheckSum:

```
dwSentenceCheckSum = wcstoul(szToken, &lpEnd, 16);
```

The function wcstoul is passed the string value to convert (szToken), a pointer to a string pointer that returns a pointer to the character in szToken that terminated the conversion (for example, a NULL character or non-numeric value), and the base used to perform the conversion (16, which is hexadecimal).

Infrared and Other Devices

The serial communications techniques described here can be used to communicate with any device that implements a suitable stream interface driver. The drivers installed on a Windows CE device are listed in the registry key \HKEY_LOCAL_MACHINE\Drivers\Active. This key contains a sub-key for each driver, and each driver is given a number (such as 01, 02, 03, and so on). The sub-key contains the name of the driver (the 'Name' value) and a 'key' value. This 'key' value contains the name of a registry key in HKEY_LOCAL_MACHINE that contains configuration data for the device. The key name for the serial communications port is usually 'drivers\builtin\serial.' This key contains information about the driver, including a 'FriendlyName' value (for example, 'Serial Cable on COM1:') and the DLL that implements the device (for example, 'Serial.Dll'). From this information you can enumerate all the serial devices present on a Windows CE device, and determine the name that should be passed to CreateFile to open the device (for example, 'COM1:').

All serial devices are accessed through calls to `CreateFile`, `ReadFile`, `WriteFile`, and `CloseHandle`. RS232 serial devices need timings and DCB settings to be configured, but other devices do not. For example, you do not need to set Baud rate or flow control settings since the driver handles transmission speed internally. The `GetCommProperties` function returns a `COMMPROP` structure for an open serial port, and this structure contains values such as the maximum Baud rate and size of the transmit and receive queues.

All devices allow parameters to be set through the `DeviceIoControl` function. This generic function allows data to be written to and obtained from the device driver. The IO control function is specified using an IoCtl code specific to a particular driver. Generally, you do not need to call `DeviceIoControl` directly, but you may find your device does publish IoCtr codes that need to be called.

Information on the infrared port is contained in the registry key `HKEY_LOCAL_MACHINE\Drivers\Builtin\IrCOMM`. The infrared port is normally mapped to a virtual communications port, for example '`COM4`'. The '`Index`' value in the `Drivers\Builtin\IrCOMM` key specifies the port number (for example, 4). Once the communications port has been determined, the infrared port can be opened using `CreateFile` for serial communications. Before the `WriteFile` and `ReadFile` functions are called, a call to `EscapeCommFunction` must be made to enable serial communications for the infrared port. This changes the infrared port from operating in '`raw`' mode to '`IrComm`' mode.

```
EscapeCommFunction(hIRPort, SETIR);
```

Once you have finished you should call `EscapeCommFunction` with `CLRIR` to return the infrared port back to raw mode.

Conclusion

This chapter has shown how to perform serial communications through a standard serial port, such as an RS232 connection. The code showed reading and writing, and controlling flow control and other communications techniques. The techniques were then applied to reading navigational information from a GPS device.

T W E L V E

Memory Management

While memory management may not rate as the most interesting subject for the majority of developers, it is important that your application use memory carefully. This is especially the case with Windows CE since devices have limited amounts of memory available to applications. Your application should allocate memory in the most appropriate way (that is from a heap, as static variables, or local variables on a stack) and ensure that memory is freed when finished with. In Windows CE, applications need also to respond to low-memory situations by carefully checking that memory allocations succeed and also by freeing up memory that is not currently essential. By doing this, an application becomes a good citizen in the Windows CE world.

Windows CE provides similar memory architecture to Windows NT/98/2000—it supports a virtual address space in which pages are mapped to physical memory. However, there are significant differences, such as the lack of a page file and the address space allocated to applications. These differences are outlined in this chapter. Just as with Windows NT/98/2000, an application can work directly with the virtual address space for memory allocations. However, the vast majority of applications can use higher-level memory allocation techniques, such as the stack and the heap.

The Virtual Address Space

In Windows CE, all applications and application data use a single 2-GB virtual address space. This is different from Windows NT/98/2000, where each application has its own 4-GB address space. The virtual address space defines the

addresses that a pointer can point at. Before data can be stored at an address, it first must be backed by physical memory.

Within the Windows CE 2-GB address space, each application is allocated a 32-MB address space into which all its memory requirements, DLLs, and code are mapped. There are 32 such address slots available, and this limitation defines the maximum number of processes that can be run in Windows CE. These 32 slots occupy 1 GB of address space, and the remaining 1 GB is used for shared memory (for example, memory-mapped files) and operating system requirements.

When a thread in a process is scheduled for execution, Windows CE moves the application down into slot 0, and effectively remaps all the addresses in the process so they fall within the range 0 to 32 MB. When the thread's execution quantum is complete, the process's addresses are remapped back into its original slot. Therefore, all addresses in a process will appear to be in the range 0 to 32 MB regardless of which slot they are assigned to. The bottom 64 KB of address space are protected and cannot be accessed by an application.

Allocating Memory for Data Storage

Before data is stored in a virtual address, data storage must be allocated to that address. In Windows NT/98/2000, data storage is allocated from the paging file, but in Windows CE data storage is allocated from the physical memory allocated to program execution. Data storage is always allocated in whole numbers of pages, and in Windows CE pages are either 1 KB or 4 KB, depending on the platform and the CPU architecture. Typically a number of pages are allocated at the same time, and these allocations must always start on an 'allocation boundary,' which in Windows CE is typically a 64-KB boundary.

Applications can manage their memory allocations at the page level using `VirtualAlloc` and `VirtualFree`. This can be a tricky business, since the page size of devices may be different. For example, if you need to allocate 18 KB of data storage, this would require 18 pages on a device with 1-KB pages and 5 pages on a device with 4-KB pages. Further, the allocation would need to start at a 64-KB allocation boundary, so either 46 KB (for a 1-KB page size device) or 44 KB (for a 4-KB page size device) of address space would remain unusable. The page size issue can be a problem even when you are targeting a single type of device—most Windows CE devices use a 1-KB page, but emulation on a desktop PC usually has a 4-KB page.

The only situation that requires direct page-level memory allocation using `VirtualAlloc` and `VirtualFree` is when an application needs to allocate a large amount of contiguous data storage. Otherwise, an application should use the heap-based allocation techniques described later in this chapter, and so avoid page size and allocation boundary issues.

Obtaining System Processor and Memory Information

The function GetSystemInfo returns information about the system processor and memory characteristics of a device in a SYSTEM_INFO structure. This function takes a single parameter that is a pointer to a SYSTEM_INFO structure. The code in Listing 12.1 shows a call to GetSystemInfo, and then code to display data relevant to Windows CE from the SYSTEM_INFO structure.

Listing 12.1 *Displaying system information using GetSystemInfo*

```
void Listing12_1()
{
  SYSTEM_INFO si;

  GetSystemInfo(&si);
  switch (si.wProcessorArchitecture)
  {
    case PROCESSOR_ARCHITECTURE_INTEL:
      cout << _T("Intel Processor");
      if(si.wProcessorLevel == 4)
        cout << _T(" 486") << endl;
      else
        cout << _T(" Pentium") << endl;
      break;
    case PROCESSOR_ARCHITECTURE_MIPS:
      cout << _T("Mips Processor");
      if(si.wProcessorLevel == 3)
        cout << _T(" R3000") << endl;
      else
        cout << _T(" R4000") << endl;
      break;
    case PROCESSOR_ARCHITECTURE_ALPHA:
      cout << _T("Alpha Processor") << endl;
      break;
    case PROCESSOR_ARCHITECTURE_PPC:
      cout << _T("PPC Processor") << endl;
      break;
    case PROCESSOR_ARCHITECTURE_SHX:
      cout << _T("SHX Processor") << endl;
      break;
    case PROCESSOR_ARCHITECTURE_ARM:
      cout << _T("ARM Processor") << endl;
      break;
    case PROCESSOR_ARCHITECTURE_IA64:
      cout << _T("IA64 Processor") << endl;
      break;
    case PROCESSOR_ARCHITECTURE_ALPHA64:
      cout << _T("Alpha 64 Processor") << endl;
      break;
```

```
    case PROCESSOR_ARCHITECTURE_UNKNOWN:
      cout << _T("Unknown Processor") << endl;
      break;
  }
  cout << _T("Processor revision: ")
    << si.wProcessorRevision << endl;
  cout << _T("Page size: ")
    << si.dwPageSize << endl;
  cout << _T("Alloc. Granularity: ")
    << si.dwAllocationGranularity << endl;
  cout << _T("Min. Application Address: ") <<
    (DWORD)si.lpMinimumApplicationAddress << endl;
  cout << _T("Max. Application Address: ") <<
    (DWORD)si.lpMaximumApplicationAddress << endl;
}
```

Typical output for a MIPS-based Windows CE device looks like the following:

```
Mips Processor R4000
Processor revision: 3154
Page size: 1024
Alloc. Granularity: 65536
Min. Application Address: 65536
Max. Application Address: 2147483647
```

The members `wProcessorArchitecture` and `wProcessorLevel` together define the type of processor the device is equipped with. The `wProcessorArchitecture` member defines the processor's architecture, such as Intel or MIPS, and `wProcessorLevel` defines the processor's level, such as R3000 or R4000. Some processors have different revisions, and this information is stored in `wProcessorRevision`. Windows CE only supports a single processor, so the `dwNumberOfProcessors` member always returns 1.

The output above shows that the page size for a MIPS device is 1 KB, and page allocations must always start on a 64-KB boundary (that is the figure returned in the `swAllocationGranularity` member). The minimum address that can be used is 64 KB, since any address below this is protected. The maximum address space is 2 GB. Remember that all processes share the same address space, so the application that produced the output above can only use up to 32 MB of address space allocated to the process.

The following output is obtained from running Listing 12.1 under emulation on a desktop PC. You can see that the address range is nearly the same but the page size is quite different. The address range is actually the address range for the process running under Windows NT, as each process is allocated a 4-GB address space; however, the upper 2 GB are protected and are reserved for the operating system. The maximum address range is actually 2 GB less the 64 KB reserved by Windows NT.

```
Intel Processor Pentium
Processor Revision 1537
Page Size: 4096
Min. Application Address: 65536
Max. Application Address: 2147418111
```

Obtaining the Current Memory Status

The function `GlobalMemoryStatus` can be used to return information about the current memory usage for Windows CE—the information is returned in a `MEMORYSTATUS` structure, as shown in Listing 12.2.

Listing 12.2 *Displaying memory usage with GlobalMemoryStatus*

```
void Listing12_2()
{
  MEMORYSTATUS ms;
  ms.dwLength = sizeof(ms);
  GlobalMemoryStatus(&ms);
  cout << _T("Total Phys: ") << ms.dwTotalPhys << endl;
  cout << _T("Avail Phys:") << ms.dwAvailPhys << endl;
  cout << _T("Total Page: ")
       << ms.dwTotalPageFile << endl;
  cout << _T("Avail Page: ")
       << ms.dwAvailPageFile << endl;
  cout << _T("Total Virtual: ")
       << ms.dwTotalVirtual << endl;
  cout << _T("Avail Virtual: ")
       << ms.dwAvailVirtual << endl;
}
```

Typical output for a Windows CE device looks like the following:

```
Total Phys: 8301568
Avail Phys: 6810624
Total Page: 0
Avail Page: 0
Total Virtual: 33554432
Avail Virtual: 29949952
```

In this case, the device has 8 MB (8301568 bytes) of memory set aside for program execution, of which around 6 MB (6810624 bytes) is available. Note that this function does not take into account the amount of memory set aside for the object store. The `dwTotalPageFile` and `dwAvailPageFile` always return 0 under Windows CE, since the operating system does not use a paging file, and data storage is allocated directly from memory. The `dwTotalVirtual`

member returns the total number of bytes of virtual address space available to the process, which is 32 MB, or 33554432 bytes. Of this, 29949952 bytes of address space are still available for use. The information returned under emulation is much the same except that the total physical and available physical memory size is always returned as 16777216, or 16 MB.

Application Memory Allocation

Applications can allocate memory for variables in one of three ways:

- Global, or static memory allocation
- Heap-based allocation
- Stack-based allocation

These three techniques allocate variables with different *scope* and *lifetime*. The variable's scope determines which part of an application can use the variable. The lifetime determines when the variable is created, and for how long. The following sections describe the three allocation techniques, the lifetime and scope of the variables, and the uses and abuses of each.

Global and Static Memory Allocation

Global variables are declared outside of functions, and static variables are declared inside functions with the 'static' modifier:

```
int g_nVar;    // global variable

void f()
{
  static int n;    // static variable
}
```

Global and static variables are created when the process starts running and are destroyed when the process terminates. The lifetime of such variables is the same as the process's lifetime. Therefore, they occupy memory for the entire time the process is running. You should *avoid using global and static variables* in Windows CE applications, as the program cannot free the memory occupied by such variables. This is especially true for global or static arrays.

A static variable's scope is the function in which it is declared. Thus, only code in the function after the static variable's declaration can access the variable. Global variables are accessible by any code in a source file that comes after the global variable's declaration, or in other source files if the source files declare the variable using the `extern` modifier.

Heap-Based Allocation

When a process is started Windows CE creates a default heap for the process. Memory can be allocated from this heap using a variety of different functions and techniques, including the following:

- The C run-time function `alloc`
- The C++ `new` operator
- The API functions `LocalAlloc` or `HeapAlloc`

Each of these techniques allows an allocation of a specified size to be made, and a pointer to the memory is returned. The memory can then be accessed through the pointer. Using a heap simplifies memory management, since you don't need to be concerned about the page allocation and de-allocation.

The heap is initially created with 384 KB of address space reserved for the heap, but without any actual physical memory associated with the heap. As memory allocations are made, physical memory is allocated to these pages. If the size of the heap exceeds 384 KB, more address space is allocated to the heap. Note that the heap may not be in contiguous memory.

Memory can be freed using one of these techniques:

- The C run-time function `free`
- The C++ `delete` operator
- The API functions `LocalFree` or `HeapFree`

The following code shows a typical allocation using `LocalAlloc` and `LocalFree`.

```
LPTSTR lpStr;
lpStr = (LPTSTR)LocalAlloc(LPTR, 100 * sizeof(TCHAR));
if(lpStr == NULL)
{
  // out of memory
}
else
{
  // use the pointer lpStr...
  LocalFree(lpStr);
}
```

The code allocates memory for 100 TCHAR characters, which under Windows CE using Unicode will result in a memory allocation of 200 bytes. The LPTR constant specifies that `LocalAlloc` will return a pointer and that the memory block will be filled with `NULL` bytes.

The space occupied by the freed block is then available to another allocation. Over time, the heap can become fragmented, so more memory and address space is used than is actually required for allocations currently in use.

This is one of the major downsides to using a heap, especially for applications that may be running over a long period of time. One solution to this problem is to create additional heaps for specific allocation purposes. These heaps can be deleted to free all the memory occupied by the heap. This technique is described later in this chapter. Note that the default heap cannot be deleted.

The scope and lifetime of data allocated from the heap is totally in the control of the application. The pointer returned from an allocation can be passed to any function, allowing the data to be accessed by those functions. However, it is generally best to limit the access to the pointer, and hence the scope, by encapsulating the pointer. This involves providing functions or a C++ class that controls access to the pointer. An application can decide when to allocate and free the memory, and therefore has control over the scope.

In general, the heap is the best place to allocate memory for variables that need a variable lifetime and must be accessed by several different functions.

Stack-Based Allocation

When a process is created, Windows CE creates a stack for the primary thread. The stack is used to store information about each function call, including any parameters passed to the function, any local variables declared in the function, and the address to where the function should return. All the information about a function call is stored in a 'stack frame.' A stack in Windows CE can be up to 60 KB, and initially a single page is allocated for the stack. Of the 60 KB, 58 KB can be used for the stack and the remaining 2 KB is used to detect stack overflows.

Any variable declared in a function, or parameter passed to a function, will use the stack for storage. When the function returns, the variables and parameters will be destroyed. The scope of variables and parameters is always the function in which they are declared. The lifetime of the variables and parameters is from the time the function is called to the time the function returns.

Each new thread created in a process must have its own stack, and Windows CE creates this automatically. Any functions called in a DLL will use the stack owned by the thread that is used to call the function.

In general, you should be careful not to declare local variables of excessive size—remember that the amount of stack used is the size of all the local variables and parameters for all function calls in the call list. Your application will fail if this exceeds 58 KB.

Creating Your Own Heaps

You should consider creating your own heap for memory allocation if you want to do either of the following:

- Make lots of memory allocations that will all be deleted at the same time
- Make lots of memory allocations of the same size

As described earlier, the default heap cannot be deleted, and so fragmentation can cause memory problems if the process executes over a long period of time. By using your own heaps, you can delete the heap periodically and therefore effectively remove fragmentation and memory wastage.

A new heap can be created using the function `HeapCreate`. This function is passed a serialization option, the initial size for the heap, and the maximum size of heap. All heaps are serialized and the initial size and maximum size are ignored. This code returns a handle to the new heap:

```
HANDLE hHeap;
hHeap = HeapCreate(0, 1024, 0);
```

Once a heap has been created, allocations can be made using the `HeapAlloc` function. For example, the following code allocates space for 100 Unicode characters and places `NULL`s in each byte.

```
LPTSTR lpStr;
lpStr = (LPTSTR) HeapAlloc(hHeap,
    HEAP_ZERO_MEMORY,
    100 * sizeof(TCHAR));
if(lpStr == NULL)
  cout << _T("Out of memory");
```

The function returns a valid handle on success, or a `NULL` if an out-of-memory condition results. All the functions that manipulate heaps, except `HeapCreate`, take a handle to the heap as the first argument. These functions can be used on the default process heap by calling the `GetProcessHeap` function to return a handle to the default process heap.

The function is passed the handle to the heap returned from `HeapCreate`, a flag (the only flag used in Windows CE is `HEAP_ZERO_MEMORY`), and the number of bytes to allocate. The number of bytes allocated may be larger than the number requested, and these extra bytes can be used by the application (although this would not be considered good practice). The `HeapSize` function can be used to return the actual size of the allocation. It is passed the handle to the heap, flags (which are always 0 with Windows CE), and the pointer to the allocation:

```
DWORD dwSize;
dwSize = HeapSize(hHeap, 0, lpStr);
```

An allocation can be freed using the `HeapFree` function, which is passed the handle to the heap, flags (0 for Windows CE), and the pointer to the allocation to be freed:

```
if(!HeapFree(hHeap, 0, lpStr))
  cout << _T("Allocation could not be freed");
```

A memory allocation can be reallocated to a different size through calling the HeapRealloc function. You need to be careful calling this function, since the reallocation may result in the memory block being moved. This results in a new pointer being returned. Finally, a heap can be deleted by calling the HeapDestroy function. You do not have to delete each individual allocation before calling this function. The HeapDestroy function is simply passed the handle to the heap:

```
if(!HeapDestroy(hHeap))
  cout << _T("Could not destroy heap");
```

Using Heaps with C++ Classes

You can use a separate heap for allocating objects for a given C++ class by overloading the new and delete operators. This is particularly useful if you are going to allocate large numbers of objects of one particular class. In the following code a C++ class called 'cHeap' is declared that has a member variable called szBuffer, and this is used to store a Unicode string. Whenever a new object of the cHeap class is allocated, the allocation for the entire class object will be made from a separate heap, and the allocation will be handled using the overloaded new and delete operators declared in the class:

```
class cHeap
{
public:
  cHeap();
  ~cHeap();
  void* operator new(size_t size);
  void operator delete(void* p);
  void putStr(LPTSTR){ wcscpy(szBuffer, pStr);}
  LPTSTR getStr() { return szBuffer;}

private:
  static HANDLE hHeap;
  static int nCount;
  TCHAR szBuffer[1024];
};
```

The static member 'hHeap' will be used to store the handle to the separate heap, and nCount records the number of instances of this heap in existence. The implementation of cHeap declares the static variables hHeap and nCount:

```
HANDLE cHeap::hHeap;
int cHeap::nCount;
```

The constructor and destructor for this class increment and decrement nCount:

```
cHeap::cHeap()
{
  nCount++;
}

cHeap::~cHeap()
{
  nCount--;
}
```

The overloaded new operator first checks whether the separate heap has been allocated, and if not, creates it. The new operator then goes on to allocate the space for the new class object using the `HeapAlloc` function. The `delete` operator frees the given class object, and if the object count is zero, deletes the heap as well.

```
void* cHeap::operator new(size_t size)
{
  if (hHeap == NULL)
  {
    hHeap = HeapCreate(0, 1024, 0);
    if(hHeap == NULL)
      cout << _T("Cannot create heap!");
  }
  return HeapAlloc(hHeap, HEAP_ZERO_MEMORY, size);
}

void cHeap::operator delete(void* p)
{
  HeapFree(hHeap, 0, p);

  if(nCount <= 0 && hHeap != NULL)
  {
    HeapDestroy(hHeap);
    hHeap = NULL;
  }
}
```

Objects of this 'cHeap' class can now be allocated with the new operator, and the memory allocation for the object will be made from the separate heap rather than from the default process heap.

```
cHeap* theObj = new cHeap();
// Use theObj pointer
delete theObj;
```

Note that if the variable is declared rather than dynamically allocated, the space used will be allocated from the stack if declared in a function:

```
cHeap myObj; // not allocated from separate heap
```

Handling Low-Memory Situations

Applications running under Windows CE should always be prepared for low- or out-of-memory situations. Requests to allocate memory may fail (in which case they return a NULL pointer). In Windows CE implementations with a shell (such as Pocket PC or Handheld PC), applications should respond to WM_HIBERNATE messages.

Windows CE recognizes three distinct low-memory threshold situations, and these are activated by any application allocating new pages of memory. With Pocket PC three low-memory situations are recognized, and the following actions are taken by the operating system:

- Hibernation. The shell sends a WM_HIBERNATE to the application that has been inactive the longest.
- Low Memory. The shell sends a WM_CLOSE message to the application that has been inactive the longest. The shell continues to send WM_CLOSE messages to applications until the free memory climbs above the low-memory threshold, or when only the foreground application remains open.
- Critical Memory. No new applications can be opened.

With Handheld PC applications are not automatically closed. Instead, an out-of-memory dialog box is displayed and the user is requested to close down applications.

The free memory values for these threshold situations depend on the platform (such as Pocket PC and Handheld PC) and the page size (either 1 KB or 4 KB). Table 12.1 shows the threshold values for Pocket PC and Handheld PC for devices with 1-KB page size. Platforms without a shell do not receive WM_HIBERNATE or WM_CLOSE messages.

Table 12.1 *Low-memory threshold values*

Threshold	Pocket PC on Windows 3.0	Handheld PC on Windows 2.11
Hibernation threshold	128 KB	200 KB
Low-memory threshold	64 KB	128 KB
Critical memory threshold	16 KB	24 KB

Responding to a WM_CLOSE Message

A Windows CE application should be prepared to receive a WM_CLOSE message from the shell and not just from the application's own interface. In response to a WM_CLOSE message from the shell, the application should save any documents without prompting the user, and free any resources prior to closing.

Responding to a WM_HIBERNATE Message

When an application receives a `WM_HIBERNATE` message, it should free up as much memory as possible by doing the following:

- De-allocating any memory structures that can be recreated
- Closing any unnecessary windows
- Deleting any fonts, menus, bitmaps, strings, or other resources

When the user or the operating system next activates the application, a `WM_ACTIVATE` message will be received by the application. At this point the application can reallocate memory or recreate windows as necessary.

Conclusion

Windows CE supports many of the same memory management techniques as desktop PC operating systems. However, Windows CE provides additional support for responding to low-memory systems that are more critical because the operating system does not use a paging file. As ever, programmers should always check for allocation failures and ensure that all allocated memory is eventually freed.

T H I R T E E N

System Information and Power Management

There are many different versions of the Windows CE operating system and platforms, so it is important that your application can determine the platform and version it is running on. Perhaps your application needs to execute a function that may or may not be present. Many of these Windows CE devices rely on battery power, so an application must be written to conserve power and also be able to monitor the current state of the battery. For example, if an application is going to initiate communications through a modem, it should determine if sufficient battery power is available.

Operating System Version Information

The function `GetVersionEx` can be used to obtain the Windows CE operating system version your application is running on. The function takes a single argument that is a pointer to a `OSVERSIONINFO` structure in which the version information is returned. The code in Listing 13.1 calls `GetVersionEx` and displays the contents of the `OSVERSIONINFO` structure.

Listing 13.1 *Obtaining operating system version information*

```
void Listing13_1()
{
  OSVERSIONINFO osVersion;

  osVersion.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
  if(!GetVersionEx(&osVersion))
```

```
void AddTask(IPOutlookApp& pOutlookApp)
{
  ITask pTask;
  LPDISPATCH lpDispatch;

  lpDispatch = pOutlookApp.CreateItem(olTaskItem);
  pTask.AttachDispatch(lpDispatch, TRUE);
  pTask.SetSubject(_T("Task created from POOM"));
  pTask.SetBody(_T("The body text for task"));
  pTask.SetImportance(olImportanceHigh);
  pTask.Save();
}
```

The `IDispatch` pointer returned from `CreateItem` is placed temporarily in the `LPDISPATCH` variable `lpDispatch`. An `ITask` class object is declared called `pTask`, and the `COleDispatchDriver::AttachDispatch` function is used to associate the `IDispatch` pointer with `pTask`. The second parameter `TRUE` passed to `AttachDispatch` specifies that the MFC classes will be responsible for reference counting.

Conclusion

This chapter has shown several mechanisms for using COM components from your applications. First, COM interfaces were accessed directly using `CoCreateInstance`, `AddRef`, `Release`, and `QueryInterface`. Next, smart pointers were used to alleviate the responsibility of managing reference counting. Automation provides a structure-based calling mechanism that allows applications a non-function-based mechanism for calling functionality in an Automation object. Code for calling `IDispatch::Invoke` directly was shown and then the MFC `COleDispatchDriver` derived classes were used to make calling Automation objects easier.

Throughout this chapter, examples showing the accessing of Pocket Outlook functionality through COM and Automation interfaces were presented. Wherever possible an application should place contact information in Pocket Outlook rather than maintaining the data in separate databases.

FIFTEEN

Microsoft Message Queue (MSMQ)

Windows CE devices are typically disconnected most of the time, but many applications still need to interact with enterprise data. Chapter 4 (Property Databases and the Registry) and Chapter 16 (ADOCE) show how data can be stored and retrieved on the device. The data, though, somehow needs to be transferred from the enterprise server onto the Windows CE device in the first place. Further, if data is updated on the Windows CE device, the changes need to be reflected back at the enterprise server. For example, a Windows CE application may allow orders to be taken, and these orders need to be transferred to a server database when the device next connects. At the same time, changes in product pricing or specification may need to be downloaded to the device.

Many desktop applications use Distributed COM (DCOM) to interact with components running under Microsoft Transaction Server (MTS or COM+), and these components implement business rule validation and access data stored in databases. However, this architecture does not work when devices are disconnected, so a different solution is required. Chapter 8 showed how HTTP and other TCP/IP protocols can be used to transfer data; however, the details of the updates need to be stored somewhere. Microsoft Message Queue (MSMQ) solves this problem by allowing applications to store messages, which will be transmitted to a server automatically upon connection. Applications running on the server can then pick up these messages and process them. Each message can contain data in any format—the application specifies the format and nature of the data each message contains.

The problem of transferring and storing data does not exist only with mobile Windows CE devices. Embedded devices may be connected to the network most of the time, but in the event of a network failure, the applications need to continue operating without the network connection. For example, consider the situation where a Windows CE embedded device is used

in a production line for testing components, and the test information is stored on a server database. If the network connection goes down, the testing application could fail and halt the entire production line. Instead, the testing results need to be stored in a queue and transmitted to the database when a network connection is present. When the network is down, the results remain in the queue waiting for the connection to be reestablished.

Overview of Microsoft Message Queue

Microsoft Message Queue (MSMQ) is a service that allows queues to be created on a computer, and for applications to write and read messages to and from the queues. Applications can access queues that are located on other computers. In the case of writing to a queue, an application can write messages to a queue located on another computer, and if the other computer is not connected, the messages are queued on the local computer. Once the computers are again connected, MSMQ automatically transfers the messages. Messages can only be read from queues on connected computers.

Any application, subject to permissions and security, can read or write messages from or to a queue. This means that a queue can be used for two-way data transmission. A queue can service requests from applications running on different computers, as each message has a sender identifier. Messages in a queue can store data in different formats. It is up to the application to interpret the format of the data. Messages have several important data items associated with them, including the following:

- Label—A textual description of the message. Useful for identifying the type of data contained in the message.
- Body—The 'payload' of the data, which can be textual or binary data of variable length.
- Time queued—A timestamp of when the message was queued.
- Time arrived—A timestamp indicating when the message was received by the queue.
- Sender identifier—Indicates the computer that sent the message.

MSMQ provides several features that are essential in a disconnected environment:

- Reliability. MSMQ provides all-or-nothing transmission of messages. Partially delivered messages will be deleted, and the original message will be resent when the next connection is made.
- Once-only delivery. MSMQ ensures that messages are not duplicated.
- In-order delivery. MSMQ ensures that messages will be read from a queue in the same order as they were written.

- Transactional support. Applications can use transactions to back out messages associated with transactions that cannot be completed. This feature is somewhat limited on Windows CE.

MSMQ uses TCP/IP sockets to provide communications and is available on Windows CE, NT, 98, and 2000. Queues use names based on Domain Name Service (DNS) names rather than IP addresses. This allows for dynamic allocation of TCP/IP addresses. As with most services, Windows CE provides a limited but useful subset of functionality found on the desktop.

There are two types of queues—public and private. Public queues are given a computer-independent name (that is, not based on a DNS or IP address). The location of the queue is resolved using Active Directory when the queue is accessed. Private queues use the DNS name of the computer and do not require Active Directory for resolution. This is faster, but means that applications need to know the physical location of a queue. Only private queues are supported in Windows CE.

Installation

Installation of MSMQ on Windows CE and desktop computers is probably more difficult than actually writing code to send and receive messages. Part of the difficulty is that MSMQ on Windows NT and Windows 2000 has an enterprise installation that configures one or more MSMQ sites. The following computers are generally used:

- A PEC (Primary Enterprise Controller), which must be installed on a Windows NT or 2000 Server installation. This computer must also run Microsoft SQL Server on Windows NT. This machine manages the MSMQ Information Store (MSMQI) used to locate message queues on other computers.
- Optional BSCs (Backup Site Controllers). These maintain backups of the MSMQI for fault tolerance purposes.
- Additional PSC (Primary Site Controllers) for managing other sites.
- Optional MSMQ Routing Servers to route messages between sites.

Other computers on the network can be independent clients (which can manage their own queues) or dependent clients (which can only access queues on other computers). All in all, setting up and managing a MSMQ installation is a job for a network administrator rather than a programmer. Most of us want to write applications using MSMQ rather than spend our time setting up numerous computers.

Windows 2000 Workstation can make this all much easier for the developer, since Windows 2000 Workstation allows a 'workgroup' installation of

MSMQ. This does not involve installing a PEC, MSC, or PSC. However, it does have two limitations:

- You cannot use public queues, only private ones.
- You must address a queue directly using its DNS (computer) name.

Given that Windows CE can only use private queues, this does not affect Windows CE MSMQ development. From personal experience, installing Windows 2000 just for MSMQ development is quicker than trying to set up a MSMQ enterprise.

Installing MSMQ on Windows CE

First, before installing MSMQ on Windows CE, you will need to change the computer name from the default the Windows CE device was shipped with (for example, `Pocket_PC`). This is to ensure that no two Windows CE devices use the same machine name for naming MSMQ queues. The name is stored in the registry in the key `HKEY_LOCAL_MACHINE\Ident\Name`. The original name of the device is stored in `HKEY_LOCAL_MACHINE\Ident\OrigName`.

Since MSMQ is not part of the standard Windows CE operating system, it needs to be installed separately for many devices, such as Pocket PC. You can do this as part of your application's installation process. In all installations, the following files need to be copied into `\windows`:

- `MSMQD.DLL`—Main MSMQ engine implemented as a device driver
- `MSMQRT.DLL`—MSMQ run-time component that implements MSMQ API
- `MSMQADM.EXE`—MSMQ administration and configuration tool

If your Windows CE device does *not* have a statically assigned IP address, you will also need to copy `NETREGD.DLL` into `\windows`. This library will register the assigned IP address and computer name with WINS (Windows Internet Naming Service). MSMQ only uses DNS (computer) names, not IP address, to reference queues. Therefore, the Windows CE device must have access to a WINS server unless the device has a statically assigned IP address. *In addition,* it must also have the following:

- Access to DNS supporting reverse lookup (that is, a DNS server that can convert an IP address to a DNS name)
- *Or* relevant DNS entries in the `LMHOSTS` file on the Windows 2000 computer and in the `HKEY_LOCAL_MACHINE\Comm\TCPIP\Hosts` registry key on the Windows CE device

Configuring DNS and IP addresses is covered later in this section. For initial development of MSMQ applications, I recommend that you use a statically assigned IP address on your Windows CE device and add LMHOST registry entries to specify the IP address and names of the computers on which you are

using queues. You will also need to add registry entries to enable MSMQ on the device. There are three ways of doing this:

- Using `VISADM.EXE`. This utility can be copied into the directory `\ProgramFiles\MSMQ`. To install MSMQ using this utility, run VISADM, click the Shortcuts button, and click 'Install.' You will then need to reboot the Windows CE device.
- Using MSMQADM. This technique would generally be used as part of your own application's installation. This application is run a number of times with various command line arguments to configure MSMQ. The section 'Installing MSMQ Using MSMQADM.EXE' in the on-line help describes this.
- Writing registry entries. This technique would be used as part of your own application's installation process and where you need to change the default installation provided by MSMQADM. The section 'Installing MSMQ Manually' in the on-line help describes the registry entries and their meanings.

The tool `VISADM.EXE` can be used to verify the installation. Run `VISADM.EXE`, click 'Shortcuts,' and click 'Verify.' This verifies that the installation is complete and lists the registry entries.

Installing MSMQ on Windows 2000

MSMQ is not part of the standard Windows 2000 installation. To install MSMQ, you will need to do the following:

- Run the Control Panel (Start menu, Settings, and Control Panel)
- Select 'Add/Remove' Programs
- Click 'Add/Remove Windows Components'
- Select 'Message Queuing Services' from the Component list and click 'Next'
- Follow through the Wizard steps

You can verify the installation using the Computer Manager application described in the next section.

Managing DNS Entries

MSMQ does not allow a computer to be referenced by IP address; you must use a computer name. For this reason DNS with reverse lookup or WINS must be available. If this is not the case (perhaps your computer is not connected to a network), you can specify entries in the Windows CE registry or in the `LMHOSTS` file on Windows 2000 to create the mapping.

On Windows CE you will need to place entries in the key `HKEY_LOCAL_MACHINE\Comm\tcpip\hosts` for each Windows 2000 computer that you want to access an MSMQ queue on. You should create a key with the name of

the Windows 2000 computer, and this key will have a value with the name 'ipaddr' containing the IP address. For example, if your Windows CE device needs to access a queue on a Windows 2000 computer called 'nickdell', you will need to add the following registry keys and values to the registry on the Windows CE device:

```
HKEY_LOCAL_MACHINE
  Comm
    Tcpip
      hosts
        nickdell
          (default)     (not set)
          ipaddr        C0 A8 37 64
```

You can see from this example that the IP address is stored as a 4-byte binary value rather than a string. The binary value 'C0 A8 37 64' represents the IP address '192.168.55.100'. The Windows CE device will need to be reset before new entries will be recognized.

If your Windows 2000 computer needs to access a queue on a Windows CE device, you will need to ensure that the Windows 2000 computer can access the Windows CE device by name (for example, 'ncg_ppc') rather than IP address. First, you can check whether the Windows CE device has registered itself using NETREGD.DLL with an available WINS or DNS server. The following steps are required:

- Run CMD.EXE on the Windows 2000 machine.
- Type the following, replacing 'ncg_ppc' with the name of your Windows CE machine:

```
ping ncg_ppc
```

If you get a message that the host is unreachable, you will need to update the LMHOSTS file on your Windows 2000 machine. This file is located, by default, in the directory \WinNT\system32\drivers\etc. Open LMHOSTS and add a line like the following at the end:

```
192.168.0.124 NCG_PPC #PRE
```

In this case '192.168.0.124' is the fixed IP address for the Windows CE device, and NCG_PPC is the Windows CE device name. You may need to restart Windows 2000 for this change to take effect.

IP Network, RAS, and ActiveSync

MSMQ requires a full IP connection between the Windows CE device and Windows NT or 2000 server to which it is connected. If you use a serial connection between a Windows CE device and desktop PC, and connect using ActiveSync, you may not be able to connect to MSMQ queues on other computers.

This is because ActiveSync does not implement a true TCP/IP connection. The problem is that many developers use an ActiveSync connection for downloading onto the device the applications they are building, and this may stop MSMQ from working.

One way around this is to configure ActiveSync to accept Network connections rather than a connection through a COM port. Then, you can configure your desktop machine to accept inbound RAS connections through the COM port. Your Windows CE device can connect using RAS (which provides a full TCP/IP connection to your network), and then the device can connect to ActiveSync through this TCIP/IP network connection.

Managing Queues on Windows 2000

Computer Management can be used to inspect queues on Windows 2000, as well as to create new queues and look at queued messages. To run Computer Management the following steps are required:

- Select the Start+Settings+Control Panel menu command.
- Double-click the 'Administrative Tools' icon.
- Double-click the 'Computer Manager' icon.
- Expand the 'Services and Applications' and then the 'Message Queuing' entries in the Tree.

Message queuing (Figure 15.1) allows four types of queue to be managed:

- Outgoing queues. This is where messages waiting to be delivered to queues on other computers are stored.
- Public queues. The location of public message queues is resolved using Active Directory. Windows CE or a Message Queue Workgroup installation on Windows 2000 does not support public queues.
- Private queues. Private message queues are accessed using the DNS name and are used by Windows CE. Figure 15.1 shows a queue that has been created called `wincequeue`.
- System queues. Journal queues are used for logging messages. The dead-letter queue receives messages that cannot be delivered.

Creating a Private Queue

Private queues can be created programmatically or, more easily, can be created using Computer Management.

- Right-click 'Private Queues' in Computer Management and select New+ Private Queue. This displays the 'Queue Name' dialog.

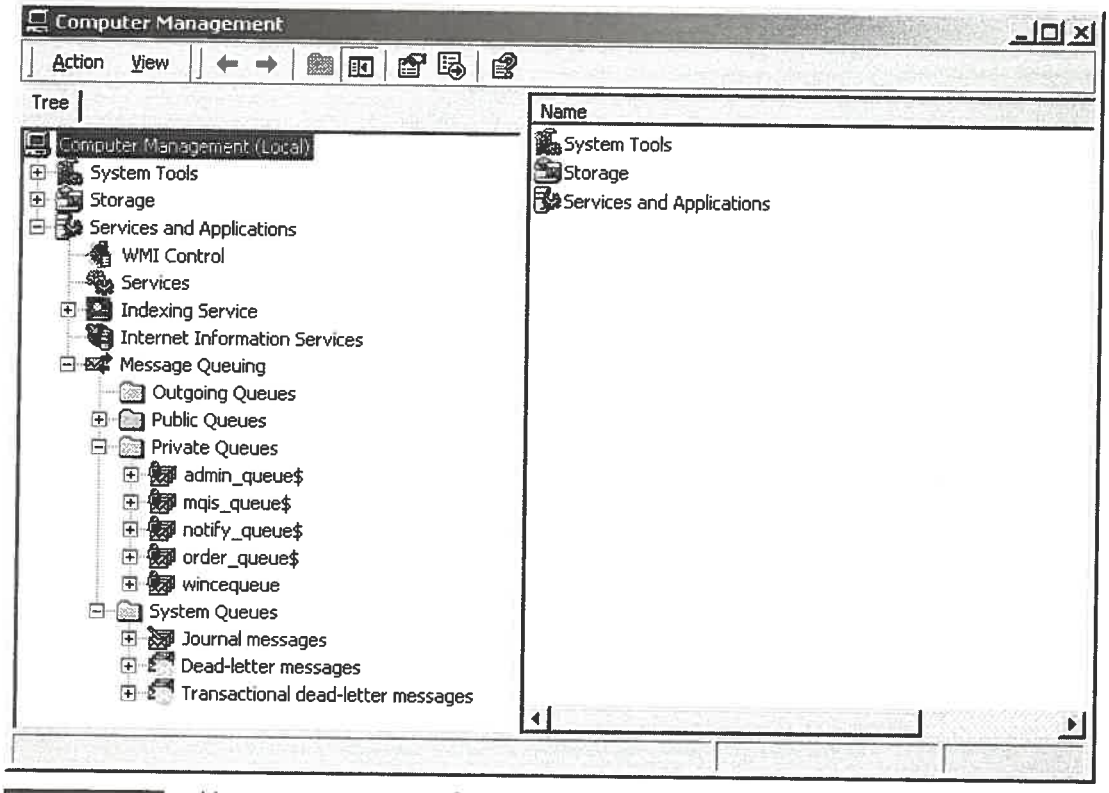


Figure 15.1 *Managing queues using Computer Management*

- Enter the name of the new queue (such as 'wincequeue' from Figure 15.1) and click OK. You have the option to create a transactional queue, described later in this chapter.

Once a private queue has been created, it can be accessed by applications running on any computer that has the appropriate security access.

The code in the next sections shows how to read messages from a queue created on a Windows 2000 computer using Visual Basic and how to write messages to the queue from a Windows CE device using C++ (Figure 15.2).

Reading Messages from a Queue in Windows 2000

The Computer Manager does not allow messages to be added to or read from a queue—you must write code to do this. You will find a Visual Basic project in the directory `\QueueServer` on the CDROM containing the code described

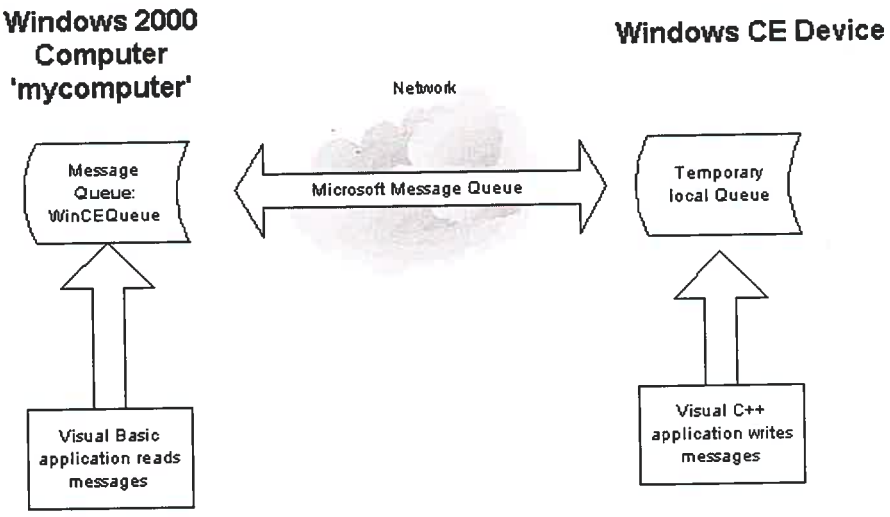


Figure 15.2 *Reading and writing a queue*

in this chapter. On Windows 2000 Visual Basic can be used with the Microsoft Message Queue object model:

- Run Visual Basic, and create a new project.
- Select the Project+References menu command. In the 'Available References' list, add a check against 'Microsoft Message Queue 2.0 Object Library.'

The code in the 'QueueServer' application opens a private queue when the form is loaded. The form has a timer control that fires an event every two seconds, and this checks to see if a new message has arrived in the queue. In the next section you will find code that runs on a Windows CE device that adds messages to this queue. Finally, the queue is closed when the form unloads:

```
Private qi As MSMQQueueInfo
Private q As MSMQQueuePrivate

Sub Form_Load()
  Set qi = New MSMQQueueInfo
  qi.PathName = ".\Private$\WinCEQueue"
  Set q = qi.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
End Sub

Private Sub Form_Unload(Cancel As Integer)
  q.Close
End Sub
```

```
Private Sub tmrMessage_Timer()
  Dim msg As MSMQMessage
  Dim s As String
  Dim sTime As String
  Dim sLabel As String
  Dim sBody As String
  Dim sSent As String

  ' check for message
  Set msg = q.Receive(ReceiveTimeout:=0)
  If (Not (msg Is Nothing)) Then
    ' have got a message
    sTime = msg.ArrivedTime
    sLabel = msg.Label
    sBody = msg.Body
    sSent = msg.SentTime
    s = s & "Sent: " & sSent & _
      " Arrived: " & sTime & ":" & _
      sLabel & vbCrLf & sBody & vbCrLf
    txtMessageLog.Text = s + txtMessageLog.Text
  End If
End Sub
```

Three Message Queue objects are used in this code:

- `MSMQQueueInfo`—Access information about an existing queue, create a new queue, or open an existing queue.
- `MSMQQueue`—Represents an open queue and allows messages to be added to the queue or read from the queue.
- `MSMQMessage`—Represents a single message to be added to the queue or read from the queue.

The queue name is specified using the '`PathName`' property. In this case a private queue called '`WinCEQueue`' on the local machine (indicated by '.') is specified. The queue is opened using the '`Open`' method. This then is passed information about how the queue is to be accessed (`MQ_RECEIVE_ACCESS` indicates that messages are to be read from the queue) and queue-sharing options (`MQ_DENY_NONE` means that other applications can open the queue for reading and writing while this application has the queue open).

The `Receive` method is called on an open queue in the timer event and checks to see if a message has arrived. This method is passed a single optional `ReceiveTimeout` parameter that specifies how long to wait before timing out. In this case the value 0 specifies no timeout value, so the call will return immediately if no message is waiting. This, together with the use of a timer, ensures that the Visual Basic application is not blocked waiting for messages. If a message is waiting, a MSMQMessage object is returned.

The MSMQMessage object is used to access the message's label, body, time sent, and time received, and this information is added to a text box. In the

next section some Windows CE code will be demonstrated that will write a message to the queue used by this Visual Basic application.

Sending Messages from Windows CE

Windows CE does not provide an object model for accessing MSMQ, so API calls need to be used. To use these API functions you will need to include `mq.h` and add the library `msmqrt.lib` to your project. A queue must first be opened, and then messages can be added to the queue. Finally, you will need to close the queue.

There are two ways in which a queue's name and location can be specified:

- Format name—A string that contains information about how the queue is named, the DNS computer on which it is located, the type of queue (for example, private or public), and the queue's name.
- Path name—A string that specifies the queue based only on the DNS, the queue type (private or public), and the queue's name.

An example of a format name to be used with Windows CE would be the following:

```
DIRECT=OS:mycomputer\Private$\WinCEQueue
```

This specifies a private queue on a computer with the DNS name '`mycomputer`', and with the queue name '`WinCEQueue`'. The '`DIRECT=OS`' specification indicates that the computer should be resolved using DNS. Other '`DIRECT`' options allow the computer to be specified by IP address, but this is not supported in Windows CE. The '`mycomputer`' DNS name is the name of the Windows 2000 computer used when creating the queue, as described in the previous section, "Reading Messages from a Queue in Windows 2000."

An example of a path name to be used with Windows CE would be as follows:

```
mycomputer\Private$\WinCEQueue
```

This specifies the same queue as the format name example above. The `MQOpenQueue` function used to open a queue requires a format name, but it is often easier to work with path names. Therefore, the `MQPathNameToFormatName` function can be used to provide a conversion:

```
TCHAR wszFormatName[256];
DWORD dwFormatNameLength = 256;

hr = MQPathNameToFormatName
     (_T("nickdell\\Private$\\WinCEQueue"),
     wszFormatName,
     &dwFormatNameLength);
```

The function MQPathNameToFormatName is passed the path name to convert (the computer 'nickdell' in this case is a Windows 2000 computer) and a string buffer in which the format name will be returned. The third parameter is a DWORD that contains the size of the buffer on calling the function and the number of characters in the format name on return.

Once the format name for the queue is obtained, the function MQOpenQueue (Table 15.1) can be called to open the queue. When opening a queue you must specify the type of access you need. For example, if you are reviewing the messages but not removing them, you can use MQ_PEEK_ACCESS. Otherwise, you may use MQ_SEND_ACCESS or MQ_RECEIVE_ACCESS to send and receive messages. A handle to the open queue is returned in a QUEUEHANDLE variable. The following code opens a queue for send access and does not deny other applications access to the queue:

```
HRESULT hr;
QUEUEHANDLE hq;

hr = MQOpenQueue(wszFormatName,
     MQ_SEND_ACCESS,
     MQ_DENY_NONE,
     &hq);
```

Table 15.1 *MQOpenQueue—Opens queue on local or remote computer*

MQOpenQueue	
LPCWSTR lpwcsFormatName	Format name of the queue.
DWORD dwAccess	Type of access required to queue:
	MQ_PEEK_ACCESS—Messages will be read but not removed from queue.
	MQ_SEND_ACCESS—Messages will be sent to the queue.
	MQ_RECEIVE_ACCESS—Messages will be read and removed from the queue.
DWORD dwShareMode	Access allowed to other applications using the queue:
	MQ_DENY_NONE—Allow other applications full access to the queue.
	MQ_DENY_RECEIVE_SHARE—Only allow other applications reading messages to access the queue.
LPQUEUEHANDLE phQueue	Pointer to a queue handle variable in which the queue handle will be returned.
HRESULT Return Value	MQ_OK on success, otherwise an error code.

The function MQSendMessage is used to send messages to an open queue. The function is passed a handle to an open queue, a pointer to a MQMSGPROPS structure describing the message to be sent, and a constant describing

the transaction options to be used. NULL for the last parameter specifies that no transactions will be used.

```
hr = MQSendMessage(hq,
     &msgprops,
     NULL);
```

Most of the work in sending messages involves forming the MQMSGPROPS structure that describes the message options and data to be sent. To send a message you will need to provide the following properties:

- PROPID_M_LABEL—A textual description describing the message. You are free to provide any textual label. This can be used by the recipient application to decide how to process the message.
- PROPID_M_BODY_TYPE—A property describing the type of data contained in the message, for example, VT_BSTR for BSTR data.
- PROPID_M_BODY—A property describing the message's data and the data itself.

To create and initialize a MQMSGPROPS, you will first need to declare a MQMSGPROPS structure. You will then need to declare a MSGPROPID array to store the property identifiers (such as PROPID_M_LABEL), a PROPVARIANT array that will contain the property data, and an optional HRESULT array used for returning error information associated with a property. The PROPVARIANT structure is used like the VARIANT structure described in Chapter 14 (COM and ActiveX). The 'vt' member contains a constant that describes the data type (such as VT_LPWSTR for a null-terminated string), and a union member that refers to the data (such as pwszVal that points to a string). In the following code, a MQMSGPROPS structure is initialized ready to store information on three properties.

```
MQMSGPROPS msgprops;
MSGPROPID aMsgPropId[3];
MQPROPVARIANT aMsgPropVar[3];
HRESULT aMsgStatus[3];
msgprops.cProp = 3;                  // Number of properties
msgprops.aPropID = aMsgPropId;   // Ids of properties
msgprops.aPropVar = aMsgPropVar;// Values of properties
msgprops.aStatus = aMsgStatus;   // Error reports
```

The PROPID_M_LABEL property contains a string for the message's label. The data type for the data stored in the MQPROPVARIANT element is therefore VT_LPWSTR, and the pwszVal member points to the string data.

```
aMsgPropId[0] = PROPID_M_LABEL;
aMsgPropVar[0].vt = VT_LPWSTR;
aMsgPropVar[0].pwszVal = _T("Test Message");
```

The PROPID_M_BODY_TYPE property describes the data type for the message's body data. The data associated with this property is an unsigned 4-byte

integer (VT_UI4), and the data in the ulVal member contains a constant describing the data type. The following code describes a message where the body data is a BSTR:

```
aMsgPropId[1] = PROPID_M_BODY_TYPE;
aMsgPropVar[1].vt = VT_UI4;
aMsgPropVar[1].ulVal = VT_BSTR;
```

Finally, the PROPID_M_BODY property describes the data for the message. The initialization depends on the type of data to be sent. The following code allocates a BSTR and sets the property to use this BSTR as the message's data:

```
BSTR bStr =
  SysAllocString(_T("Body text for the message"));
aMsgPropId[2] = PROPID_M_BODY;
aMsgPropVar[2].vt = VT_VECTOR|VT_UI1;
aMsgPropVar[2].caub.pElems = (LPBYTE)bStr;
aMsgPropVar[2].caub.cElems =
        SysStringByteLen(bStr);
```

The data type VT_VECTOR | VT_UI1 specifies that the data is to be passed as a counted array (that is, an array with a given size). The caub.pElems member describes the length of the data, and caub.cElems points to the data itself. The code in Listing 15.1 shows opening a queue, initializing the properties, and sending the message using MQSendMessage. Finally, the queue is closed through a call to MQCloseQueue—this function takes a single parameter that is the handle to the queue to close. This code will send a message to a queue that can be read by the Visual Basic code described in the section "Reading Messages from a Queue in Windows 2000."

Listing 15.1 *Opening queue and sending a message*

```
#include <mq.h>
// Add MSMQRT.LIB to project

void DisplayOpenError(HRESULT hr)
{
  if(hr == MQ_ERROR_ACCESS_DENIED)
      cout << _T("Don't have access rights") << endl;
  else if(hr == MQ_ERROR_ILLEGAL_FORMATNAME)
       cout << _T("Illegal Format Name") << endl;
  else if(hr == MQ_ERROR_QUEUE_NOT_FOUND )
       cout << _T("Queue not found") << endl;
  else if(hr == MQ_ERROR_SERVICE_NOT_AVAILABLE )
       cout << _T("Cannot connect to queue mgr")
            << endl;
  else if(hr == MQ_ERROR_INVALID_PARAMETER )
       cout << _T("Invalid Parameter") << endl;
  else if(hr == MQ_ERROR_SHARING_VIOLATION )
       cout << _T("Sharing violation") << endl;
```

```
    else if(hr == MQ_ERROR_UNSUPPORTED_ACCESS_MODE )
          cout << _T("Invalid access mode") << endl;
    else if(hr ==
          MQ_ERROR_UNSUPPORTED_FORMATNAME_OPERATION)
          cout << _T("Invalid format name") << endl;
    else
          cout << _T("Unexpected Error") << endl;
}

void Listing15_1()
{
    HRESULT hr;
    QUEUEHANDLE hq;

    TCHAR wszFormatName[256];
    DWORD dwFormatNameLength = 256;

    hr = MQPathNameToFormatName
         (_T("nickdell\\Private$\\WinCEQueue"),
         wszFormatName,
         &dwFormatNameLength);
    cout << wszFormatName << endl;

    hr = MQOpenQueue(wszFormatName,
         MQ_SEND_ACCESS,
         MQ_DENY_NONE,
         &hq);
    if(hr == MQ_OK)
      cout << _T("Opened queue") << endl;
    else
    {
      DisplayOpenError(hr);
      return;
    }

    DWORD cPropId = 0;

    MQMSGPROPS msgprops;
    MSGPROPID aMsgPropId[4];
    MQPROPVARIANT aMsgPropVar[4];
    HRESULT aMsgStatus[4];

    aMsgPropId[cPropId] = PROPID_M_LABEL;
    aMsgPropVar[cPropId].vt = VT_LPWSTR;
    aMsgPropVar[cPropId].pwszVal = _T("Test Message");
    cPropId++;

    aMsgPropId[cPropId] = PROPID_M_BODY_TYPE;
    aMsgPropVar[cPropId].vt = VT_UI4;
    aMsgPropVar[cPropId].bVal = VT_BSTR;
    cPropId++;

    BSTR bStr = SysAllocString(
         _T("Body text for the message"));
```

```
    aMsgPropId[cPropId] = PROPID_M_BODY;
    aMsgPropVar[cPropId].vt = VT_VECTOR|VT_UI1;
    aMsgPropVar[cPropId].caub.pElems = (LPBYTE)bStr;
    aMsgPropVar[cPropId].caub.cElems =
        SysStringByteLen(bStr);
    cPropId++;

    msgprops.cProp = cPropId;
    msgprops.aPropID = aMsgPropId;
    msgprops.aPropVar = aMsgPropVar;
    msgprops.aStatus = aMsgStatus;

    hr = MQSendMessage(hq,
          &msgprops,
          NULL);
    if (FAILED(hr))
      cout << _T("Could not send message") << endl;

    else
      cout << _T("Message queued") << endl;
    MQCloseQueue(hq);
}
```

If the Windows CE device on which this code runs cannot access the Windows 2000 machine where `WinCEQueue` is located, the messages will be stored in a local temporary queue. When the queue can next be accessed (for example, when the Windows CE device connects using RAS), MSMQ will automatically transfer the messages to the queue.

Creating a New Queue

New queues can be created on a Windows CE device by calling the `MQCreateQueue` (Table 15.2) function and initializing a `MQQUEUEPROPS` structure with the following properties:

- `PROPID_Q_PATHNAME`—Pathname for the new queue
- `PROPID_Q_LABEL`—Label (or description) for the new queue

The data for both these properties is `VT_LPWSTR`. The `pwszVal` member for `PROPID_Q_PATHNAME` is a pointer to a string containing the pathname. In the following code example, the '.' refers to the local Windows CE device, 'PRIVATE$' specifies this is a private queue (remember, public queues are not supported), and 'WinCEInQueue' is the name of the new queue.

```
LPWSTR wszPathName = _T(".\\PRIVATE$\\WinCEInQueue");
aQueuePropId[0] = PROPID_Q_PATHNAME;
aQueuePropVar[0].vt = VT_LPWSTR;
aQueuePropVar[0].pwszVal = wszPathName;
```

Table 15.2 *MQCreateQueue—Creates a new queue*

MQCreateQueue	
PSECURITY_DESCRIPTOR pSecurityDescriptor	Not supported, pass as NULL.
MQQUEUEPROPS *pQueueProps	Pointer to a MQQUEUEPROPS structure describing the queue to create.
LPWSTR lpwcsFormatName	Pointer to a buffer to receive the format name of the new queue. This can be NULL.
LPDWORD lpdwFormatNameLength	Length of the new buffer receiving the format name.
HRESULT Return Value	MQ_OK for success, otherwise error code.

The PROPID_Q_LABEL can be used to provide a more descriptive name for the queue:

```
LPWSTR wszQueueLabel =
  _T("Message to be received by Windows CE Device");

aQueuePropId[1] = PROPID_Q_LABEL;
aQueuePropVar[1].vt = VT_LPWSTR;
aQueuePropVar[1].pwszVal = wszQueueLabel;
```

The queue to be created is specified by a pathname, and the MQCreateQueue function will return the format name if required. The code in Listing 15.2 shows how to initialize properties and call MQCreateQueue to create a new queue on a Windows CE Device.

Listing 15.2 *Creating a new queue*

```
void Listing15_2()
{
  DWORD cPropId = 0;

  MQQUEUEPROPS QueueProps;
  MQPROPVARIANT aQueuePropVar[2];
  QUEUEPROPID aQueuePropId[2];
  HRESULT aQueueStatus[2];

  HRESULT hr
  PSECURITY_DESCRIPTOR pSecurityDescriptor=NULL

  // Queue pathname
  LPWSTR wszPathName = _T(".\\PRIVATE$\\WinCEInQueue");
  // Queue label
  LPWSTR wszQueueLabel =
    _T("Message to be received by Windows CE Device");
```

```
// Format name buffer for queue
DWORD dwFormatNameLength = 256
WCHAR wszFormatName[256];

aQueuePropId[cPropId] = PROPID_Q_PATHNAME;
aQueuePropVar[cPropId].vt = VT_LPWSTR;
aQueuePropVar[cPropId].pwszVal = wszPathName;
cPropId++;

aQueuePropId[cPropId] = PROPID_Q_LABEL;
aQueuePropVar[cPropId].vt = VT_LPWSTR;
aQueuePropVar[cPropId].pwszVal = wszQueueLabel;
cPropId++;

QueueProps.cProp = cPropId;
QueueProps.aPropID = aQueuePropId;
QueueProps.aPropVar = aQueuePropVar;
QueueProps.aStatus = aQueueStatus;

hr = MQCreateQueue(pSecurityDescriptor,
    &QueueProps,
    wszFormatName,
    &dwFormatNameLength);

if(hr == MQ_OK)
      cout << wszFormatName << _T(" created") << endl;
else if(hr == MQ_ERROR_ACCESS_DENIED )
      cout << _T("Access Denied") << endl;
else if(hr == MQ_ERROR_ILLEGAL_PROPERTY_VALUE )
      cout << _T("Illegal Property Value") << endl;
else if(hr == MQ_ERROR_ILLEGAL_QUEUE_PATHNAME )
      cout << _T("Illegal pathname") << endl;
else if(hr == MQ_ERROR_ILLEGAL_SECURITY_DESCRIPTOR )
      cout << _T("Illegal security descriptor")
           << endl;
else if(hr == MQ_ERROR_INSUFFICIENT_PROPERTIES )
      cout << _T("Path name not specified") << endl;
else if(hr == MQ_ERROR_INVALID_OWNER )
      cout << _T("Invalid owner") << endl;
else if(hr == MQ_ERROR_PROPERTY )
      cout << _T("Error in property specification")
           << endl;
else if(hr == MQ_ERROR_PROPERTY_NOTALLOWED )
      cout <<
      _T("Property not allowed when creating queue")
           << endl;
else if(hr == MQ_ERROR_QUEUE_EXISTS )
      cout << _T("Queue already exists") << endl;
else if(hr == MQ_ERROR_SERVICE_NOT_AVAILABLE )
      cout << _T("Service not available") << endl;
else if(hr ==
      MQ_INFORMATION_FORMATNAME_BUFFER_TOO_SMALL )
```

```
        cout << _T("Format name buffer too small")
        << endl;
  else if(hr == MQ_INFORMATION_PROPERTY )
        cout <<
        _T("Succeeded, but property returned warning")
             << endl;
}
```

Once the queue has been created, you will need to open the queue before messages can be sent or received from it. Queues can be deleted by calling the `MQDeleteQueue` function, and this is passed the format name of the queue to be deleted.

Reading Messages from a Queue

Messages can be read from a queue on the same Windows CE device or on another computer. You need to have a valid network connection to the other computer to read from a remote queue. To read one or more messages from a queue, you must do the following:

- Open the queue using `MQOpenQueue`
- Initialize a `MQMSGPROPS` structure in which the message will be received
- Call `MQReceiveMessage` (Table 15.3) to read a message, if one is present
- Close the queue when finished reading messages by calling `MQCloseQueue`

The minimum properties needed to pass to `MQReceiveMessage` are the following:

- `PROPID_M_BODY_SIZE`—Property receives the number of bytes in the message body
- `PROPID_M_BODY`—Property receives the message body data

The `PROPID_M_BODY_SIZE` property is initialized as shown in the following code fragment. After a successful call to `MQReceiveMessage`, the `aMsgPropVar[0].ulVal` member will contain the number of bytes in the message body.

```
aMsgPropId[0] = PROPID_M_BODY_SIZE;
aMsgPropVar[0].vt = VT_UI4;
```

You will need to allocate a buffer in which the message body will be received, and initialize the `PROPID_M_BODY` property with this pointer. In the following code, a 1-KB buffer is allocated, and the pointer is assigned to the `pElems` member. The size of the buffer is assigned to the `cElems` member.

Table 15.3 *MQReceiveMessage—Reads a message from the queue*

MQReceiveMessage	
QUEUEHANDLE hSource	Handle to an open queue.
DWORD dwTimeout	Timeout to wait for message, INFINITE to wait forever, or 0 to return immediately if no message is present.
DWORD dwAction	How to access the queue: MQ_ACTION_RECEIVE—Read the next message and remove message. MQ_ACTION_PEEK_CURRENT—Read current message, but do not remove it. MQ_ACTION_PEEK_NEXT—Use a cursor to read the next message, but do not remove it.
MQMSGPROPS pMessageProps	Structure in which the message will be received.
LPOVERLAPPED lpOverlapped	Pointer to an OVERLAPPED structure for asynchronous message reading. Use NULL for synchronous access.
PMQRECEIVECALLBACK fnReceiveCallback	Pointer to callback function for asynchronous message reads. Use NULL for synchronous access.
HANDLE hCursor	Handle to cursor for reading messages, or NULL for no cursor.
Transaction *pTransaction	Not supported, pass as NULL.
HRESULT Return Value	MQ_OK for success, or error message on failure.

```
DWORD dwBodyBufferSize = 1024;
LPTSTR lpszBodyBuffer = new TCHAR[dwBodyBufferSize];
aMsgPropId[1] = PROPID_M_BODY;
aMsgPropVar[1].vt = VT_VECTOR|VT_UI1;
aMsgPropVar[1].caub.pElems =
      (UCHAR*)lpszBodyBuffer;
aMsgPropVar[1].caub.cElems = dwBodyBufferSize;
```

The code in Listing 15.3 shows opening the queue on the Windows CE device created in Listing 15.2, and reading a message from the queue. The timeout of 0 means that MQReceiveMessage will return immediately with a message if one is present, or return a MQ_ERROR_IO_TIMEOUT error if none is present. Since MQReceiveMessage is called on the primary thread, it is important that the call to MQReceiveMessage does not block for any length of time. The code displays the number of bytes in the message body and then displays the contents of the body. Since the receive action is MQ_ACTION_RECEIVE, the message will be removed from the queue once it has been read.

Listing 15.3 *Reading a message from a queue*

```
void DisplayReadError(HRESULT hr)
{
  if(hr == MQ_ERROR_ACCESS_DENIED)
        cout << _T("Don't have access rights") << endl;
  else if(hr == MQ_ERROR_BUFFER_OVERFLOW )
        cout << _T("Buffer Overflow") << endl;
  else if(hr == MQ_ERROR_SENDERID_BUFFER_TOO_SMALL )
        cout << _T("Sender ID Buffer too small") << endl;
  else if(hr == MQ_ERROR_SYMM_KEY_BUFFER_TOO_SMALL )
        cout << _T("Symmetric key buffer too small")
             << endl;
  else if(hr == MQ_ERROR_SENDER_CERT_BUFFER_TOO_SMALL )
        cout << _T("Cert buffer too small") << endl;
  else if(hr == MQ_ERROR_SIGNATURE_BUFFER_TOO_SMALL )
        cout << _T("Signature buffer too small") << endl;
  else if(hr == MQ_ERROR_PROV_NAME_BUFFER_TOO_SMALL )
        cout << _T("Provider name too small") << endl;
  else if(hr == MQ_ERROR_LABEL_BUFFER_TOO_SMALL)
        cout << _T("Label buffer too small") << endl;
  else if(hr == MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL )
        cout << _T("Format name buffer too small")
             << endl;
  else if(hr == MQ_ERROR_DTC_CONNECT )
        cout << _T("Cannot connect to DTC") << endl;
  else if(hr == MQ_ERROR_INSUFFICIENT_PROPERTIES )
        cout << _T("Insufficient properties") << endl;
  else if(hr == MQ_ERROR_INVALID_HANDLE )
        cout << _T("Invalid queue handle") << endl;
  else if(hr == MQ_ERROR_IO_TIMEOUT )
        cout << _T("Timeout") << endl;
  else if(hr == MQ_ERROR_MESSAGE_ALREADY_RECEIVED )
        cout << _T("Message has been removed from queue")
             << endl;
  else if(hr == MQ_ERROR_OPERATION_CANCELLED )
        cout << _T("Operation cancelled") << endl;
  else if(hr == MQ_ERROR_PROPERTY )
        cout << _T("Property error") << endl;
  else if(hr == MQ_ERROR_QUEUE_DELETED )
        cout << _T("Queue deleted") << endl;
  else if(hr == MQ_ERROR_ILLEGAL_CURSOR_ACTION )
        cout << _T("Illegal cursor action") << endl;
  else if(hr == MQ_ERROR_SERVICE_NOT_AVAILABLE )
        cout << _T("Service not available") << endl;
  else if(hr == MQ_ERROR_STALE_HANDLE )
        cout << _T("Stale handle") << endl;
  else if(hr == MQ_ERROR_TRANSACTION_USAGE )
        cout << _T("Transaction Error") << endl;
```

```
SELECT * FROM Orders JOIN OrderDetails
    ON (Orders.OrderNum = OrderDetails.OrderNum)
```

The code in Listing 16.13 opens a recordset on this SELECT statement to return all the orders and related OrderDetails records. The opening of the recordset is very similar to Listing 16.6. The code to display the contents of the recordset is generic—it can list the field names and values for any fields collection passed into it. A 'for' loop is used to iterate across all the fields in the fields collection, using the GetItem and an integer index to obtain a pointer to each field. The name of the field is obtained through the 'Name' property, and the value from the GetValue function. GetValue will return a VARIANT with the vt value containing an appropriate value for the underlying field in the table (such as VT_I4, VT_DATE, and so on). Since the data is to be displayed, the easiest thing to do is convert the VARIANT to a BSTR regardless of the original data type. The _variant_t class member 'ChangeType' can do this, as follows:

```
varValue.ChangeType(VT_BSTR, NULL);
```

This function is passed the data type to convert the VARIANT to and a second parameter specifying where the converted VARIANT should be placed. Passing NULL specifies that the conversion should take place in situ, and the original variant value is replaced by the newly converted value.

Listing 16.13 *The SELECT with JOIN statement*

```
void DisplayOrders(AdoNS::FieldsPtr & pFields)
{
  AdoNS::FieldPtr pField;
  _variant_t varValue, varIndex, varStringValue;
  _bstr_t bstrIndex;

  for(short i = 0; i < pFields->Count; i++)
  {
    varIndex = i;
    pField = pFields->GetItem(varIndex);
    cout << (LPTSTR)pField->Name << _T(":");
    varValue = pField->GetValue();
    varValue.ChangeType(VT_BSTR, NULL);
    cout << varValue.bstrVal << _T(" ");
  }
  cout << endl;
}

void Listing16_13()
{
  HRESULT hr;
  AdoNS::_RecordsetPtr pRecordset;
```

```
    _bstr_t bstrConnection(lpConnection);
    _variant_t varConnection(bstrConnection);
    _bstr_t bstrQuery(_T("SELECT * FROM Orders \
      JOIN OrderDetails \
      ON (Orders.OrderNum = OrderDetails.OrderNum)"));
    _variant_t varQuery(bstrQuery);

    hr = pRecordset.CreateInstance
        (_T("ADOCE.Recordset.3.1"));
    if(FAILED(hr))
    {
      cout << _T("Could not create recordset:")
           << hr << endl;
      return;
    }
    // Open the base table and retrieve rows
    //
    hr = pRecordset->Open(varQuery,
          varConnection,
          AdoNS::adOpenStatic,
          AdoNS::adLockReadOnly,
          AdoNS::adCmdText);
    if(FAILED(hr))
    {
      cout << _T("Could not open recordset") << endl;
      return;
    }

    while(!pRecordset->GetA_EOF())
    {
      AdoNS::FieldsPtr pFields;
      pFields = pRecordset->GetFields();
      DisplayOrders(pFields);
      pRecordset->MoveNext();
    }
    pRecordset->Close();
}
```

Error Handling

In the code shown so far in this chapter, errors trapped by the smart pointer wrapper functions have resulted in `_com_issue_errorex` being called. This has displayed the `HRESULT` generated by the offending call. The problem, though, is that some of the smart pointer wrapper functions attempt to continue execution and use invalid interface pointers. For example, here is the wrapper function for `Connection::Execute`:

```
inline _RecordsetPtr _Connection::Execute (
    _bstr_t CommandText,
    VARIANT * RecordsAffected, long Options )
{
  struct _Recordset * _result;
  HRESULT _hr = raw_Execute(CommandText,
      RecordsAffected, Options, &_result);
  if (FAILED(_hr))
    _com_issue_errorex(_hr, this, __uuidof(this));
  return _RecordsetPtr(_result, false);
}
```

You can see that a `_RecordsetPtr` is created from `_result` even if an error was detected and `_com_issue_errorex` is called. Any code you have after calling `Execute` will probably not be executed, as this will generate a memory exception fault. One solution is to call the `raw_` versions of the functions (like `raw_Execute` in the above code), since these will always return `HRESULT` values to your code. You will then need to create the smart pointer class objects (such as `_RecordsetPtr`) from the interface pointers (such as `Recordset`).

Another solution is to use exception handling. Unfortunately, you cannot use C++ exception handling since it is not supported on Windows CE. Consequently, you will need to deal with Win32 Structured Exception Handling (SEH). This is a large topic, and the examples shown here are simple and only show rudimentary use of SEH.

First, you will need to raise an error in the function `_com_issue_errorex`. Calling the Windows CE function `RaiseException` does this. The first argument is the error code, and in this case the `HRESULT` value that caused the problem is used. The other parameters concern flags and passing additional exception information, and these are not used here.

```
void _com_issue_errorex(HRESULT hr, IUnknown* pUnkn,
      REFIID riid)
{
  RaiseException(hr, 0, 0, NULL);
}
```

Next, you will need to trap the exception in your code using '`__try`' and '`__except`' blocks. In Listing 16.14 a connection is made, and then an obviously bad SQL statement is executed through that connection in a `__try` block. This will result in `_com_issue_errorex` being executed and an exception being generated. Execution will jump to the `__except` block. The error code is obtained using the `GetExceptionCode` Windows CE function. This must be executed in brackets following the `__except` statement. The `HRESULT` is then displayed to the user. Note that the code following the `__except` statement (the `Close` connection) will be executed, so the connection will be closed cleanly. Without the exception handling, the connection would be left open. This can cause problems for subsequent database access calls.

Listing 16.14 *Structured exception handling*

```
void Listing16_14()
{
  AdoNS::_ConnectionPtr pConnection;
  HRESULT hr;
  EXCEPTION_RECORD ExceptionRecord;

  if(!GetConnection(pConnection))
    return;
  _bstr_t bStrSQL(_T("BAD SQL Command "));

  __try
  {
    ExecuteSQL(pConnection, bStrSQL);
  }
  __except (hr = GetExceptionCode(),
        EXCEPTION_EXECUTE_HANDLER)
  {
    cout << _T("Trapped Failure: ") << hr << endl;
  }
  pConnection->Close();
  cout << _T("Finished") << endl;
}
```

During the course of your ADOCE and ADOXCE programming exploits, you will encounter many different `HRESULT` errors. These can either be returned from ADO or ADOXCE, or from the OLEDB provider for the database you are using. I suggest you search the MSDN Library that is shipped with Microsoft Visual Studio (rather than Microsoft eMbedded Visual C++) for the error number. You are likely to find a description of the error there.

Transactions

There are many situations where a number of SQL statements must be executed. It is imperative that all of these statements succeed or, if one fails, that the changes made by other statements are removed from the database. This is important to ensure data integrity. For example, in the case where the `INSERT` statement was used to add a new order, consisting of several `OrderDetails` records, all the records should be added to the database or, if one insertion fails, the other records should be removed. This can be achieved by using transactions.

Executing `BeginTrans` through a Connection interface starts a transaction. The SQL statements can then be executed through that same Connection interface. Once complete, the application can call `CommitTrans` if all com-

pleted successfully, or `RollbackTrans` to backout any changes made from the time the `BeginTrans` was executed.

For example, the following two SQL statements will delete all the `Orders` and `OrderDetail` records. These statements should be in a transaction, since all the information needs to be deleted, or none.

```
DELETE FROM Orders
DELETE FROM OrderDetails
```

Listing 16.15 shows a transaction placed around these two `DELETE` statements, together with exception handling. If an exception is detected, a `ROLLBACK` is executed. If no exception occurs the changes are committed to the database.

Listing 16.15 *Transactions*

```
void Listing16_15()
{
  AdoNS::_ConnectionPtr pConnection;
  HRESULT hr;
  EXCEPTION_RECORD ExceptionRecord;

  if(!GetConnection(pConnection))
    return;
  _bstr_t bStrSQL;

  __try
  {
    pConnection->BeginTrans();
    bStrSQL = _T("DELETE FROM Orders");
    ExecuteSQL(pConnection, bStrSQL);
    bStrSQL = _T("DELETE FROM OrderDetails");
    ExecuteSQL(pConnection, bStrSQL);
    pConnection->CommitTrans();
  }
  __except (hr = GetExceptionCode(),
        EXCEPTION_EXECUTE_HANDLER)
  {
    cout << _T("Trapped Failure: ") << hr << endl;
    pConnection->RollbackTrans();
  }
  pConnection->Close();
  cout << _T("Finished") << endl;
}
```

Conclusion

This chapter has shown how to use ADOCE and ADOXCE to access databases on Windows CE, specifically Microsoft SQL Server for Windows CE. Databases

can be created, and tables, fields (columns), and indexes can be added either using the ADOXCE object model or through SQL DDL statements. Data can be added to and extracted from the database through recordsets and SQL DML statements. There is a lot more to ADOCE and ADOXCE than is shown in the chapter. Take a look at the generated `.tlh` and `.tli` files for the smart pointer classes. You can use the Microsoft Visual Studio documentation to help work out what the functions do. Using Microsoft SQL Server for Windows CE is faster than using a property database, especially for larger amounts of data, and makes manipulating relational data much more efficient and reliable.

SEVENTEEN

ActiveSync

ActiveSync facilitates synchronization of data between a desktop and a companion application running on a Windows CE device. Users expect an application to automatically transfer data to and from the Windows CE device and to synchronize changes, so wherever applicable you should implement ActiveSync functionality in your applications. However, this is one of the most difficult tasks you are likely to encounter in Windows CE development.

You will need to know about Component Object Model (Chapter 14, COM and ActiveX), CE property database programming (Chapter 4), writing Dynamic Link Libraries, registry manipulation (Chapter 4), and process and thread synchronization (Chapter 6). Adding ActiveSync functionality is one of those annoying programming tasks where you cannot see something working until you have implemented lots of code both on the desktop and the Windows CE device.

ActiveSync 3.1 replaces Windows CE Services 2 and improves reliability, setup, and installation and improves performance. ActiveSync 3.1 does not require configuration or installation changes on the Windows CE device. You can write ActiveSync code that will also run with Windows CE Services. Windows CE Services for Windows CE 2.11 and ActiveSync 3.1 provide support for database volumes that is not provided in earlier versions.

You will have experienced the benefits of ActiveSync with the Pocket Outlook Applications, such as automatic synchronization of appointments, contact information, and tasks. If you are writing a companion application for Windows CE that shares data with your desktop application, you will need to implement ActiveSync. You implement an ActiveSync Service Provider, and ActiveSync provides the service manager.

You can implement manual synchronization (which occurs when the device connects or when the user clicks the "Synchronize" button in ActiveSync),

or continuous synchronization (with automatic, instantaneous updates). The latter takes more effort to implement, primarily on the desktop.

ActiveSync Items, Folders, and Store

First, you need to understand how ActiveSync organizes data in items, folders, and the store.

Item

The basic unit of synchronization is the item. In Pocket Outlook, an appointment or contact is an item. Each item has two important pieces of information associated with it:

1. A unique field identifier. The identifier for an item should never change and should be unique. Identifiers for deleted items should not be reused. Further, the identifier should be ordered—that is, the identifier can be used to determine if an item comes before or after another item. The identifier could be the timestamp of when the object was created.
2. A value used to determine if the item has changed. This could be the timestamp of when the object was last modified.

You can define these data items in any way you choose, but you should keep them as small as possible. ActiveSync stores a copy of the data items in the file `repl.dat` for each item being synchronized. There is a `repl.dat` file for each profile on the desktop PC.

You are free to define the size and nature of these two pieces of data. You communicate this data to ActiveSync through the generic pointer type `HREPLITEM`. Note that `HREPLITEM` structures are used and stored only on the desktop PC, not on the Windows CE device.

Folder

Items are stored in folders. Folders group items of a similar type. For example, you might have a folder for appointments and a folder for contacts. You can use any data you like to identify the folder, and this data is passed to ActiveSync through the generic pointer type `HREPLFLD`. These structures are used only on the desktop PC and not on the Windows CE device.

Folders are a way to group items together logically. ActiveSync makes no stipulations as to how or where folders are stored. If possible, use a single folder since it makes programming simpler.

Store

Folders are organized into a single store. Each store has a unique string identifier that is used to link the provider on the device to the provider on the desktop. This identifier is a COM progid, such as "MS.WinCE.Outlook". You will implement a DLL for the device and another for the desktop PC that will support synchronization for the store.

Any storage technique can be used for data in the store, but the following will make for an easier implementation:

1. Use a single CE property database on the CE device. This will ensure proper synchronization of updates and implement continuous synchronization automatically. Use a single record for each item.
2. Use a database (such as Microsoft SQL Server or Access) on the desktop. You can use flat files, but take care to implement synchronization (such as an event, see Chapter 6) to ensure that the user and ActiveSync do not attempt to update the file simultaneously.

Steps to Implement Device Synchronization

Follow these steps to implement device synchronization:

1. Create a standard DLL project.
2. Write code to register the device ActiveSync provider in the registry.
3. Implement the ActiveSync `IReplObjHandler` COM interface. This interface implements functions to take your items and convert them to a stream of bytes (serialization) and vice versa (deserialization).
4. Implement the following exported functions that will be called by ActiveSync:

   `InitObjType`—Called by ActiveSync when the service is loaded and unloaded.

   `ObjectNotify`—Called by ActiveSync when the item in the store is added, deleted, or updated. The function returns `TRUE` if the item is to be synchronized.

   `GetObjTypeInfo`—Called by ActiveSync to obtain information about the object store, which is typically a CE property database.
5. Write code to add, update, and delete items from the store. This code can typically be shared with the application that will need to perform the same tasks.

The DLL will need to implement the `IReplObjectHandler` COM interface but does not need to be a fully implemented COM component. This means that a class factory and the standard exported functions (such as `DllCanUnloadNow`) do not need to be implemented.

Steps to Implement Desktop Synchronization

Follow these steps to implement basic desktop synchronization:

1. Create a standard DLL project.
2. Write code to register the desktop ActiveSync provider in the registry.
3. Implement the ActiveSync `IReplObjHandler` COM interface. This interface is the same one as implemented in the device DLL.
4. Implement the ActiveSync `IReplStore` COM interface. This interface implements functions to manage the store, folders, and items; manage conflicts; remove duplicates; and present user-interface dialogs to set options.
5. Decide on the data used for `HREPLFLD` (folder identifiers) and `HREPLITEM` (item identifiers). For reasons that will become apparent later, it is easiest to define a structure with a union defining the data for the folder and field identifier.
6. Write code to add, update, and delete items from the store. This code typically can be shared with the application that will need to perform the same tasks.

Unlike the device DLL, the desktop DLL needs to implement a true COM component. This means that a class factory and standard COM-exported functions are required. The DLL can be written from the ground up (as is done with the sample application presented in this chapter), or you can choose to use MFC or ATL to simplify the task.

The Windows CE DLL will obviously be implemented using Unicode (wide) strings. The desktop PC DLL is best implemented to use ANSI (multibyte) characters, since the structures passed from the ActiveSync service contain ANSI strings. The data transferred between the Windows CE and desktop PC DLLs can be either Unicode or ANSI—it is your choice. However, you will need to convert the strings from Unicode to ANSI (for data being transferred from the Windows CE device to the desktop PC) or from ANSI to Unicode (for data being transferred from the desktop PC to the Windows CE device). You can perform this conversion either on the CE device or on the desktop PC.

Additional Steps for Continuous Synchronization

You need to implement two extra bits of code if you want synchronization to occur continuously while the Windows CE device remains connected:

1. Call appropriate functions in the `IReplNotify` interface provided by ActiveSync to notify changes to items in the store.
2. Write synchronization codes to allow the desktop application to notify your ActiveSync that item changes have occurred.

The Sample Application

The accompanying CDROM contains a sample application that illustrates the implementation of a simple ActiveSync provider. The application synchronizes items with a single string of up to 256 characters. The items are stored in a Windows CE property database on the device and a flat file on the desktop PC. The source code is located in the directory `\ActiveSync`. The application consists of the following projects:

1. **`CIDevice`**—A Windows CE MFC application that presents a simple user interface to manipulate the records in the database, located in `\ActiveSync\asdevice\cldesktop`. This application is a straightforward MFC application, not described here, that manipulates a Windows CE property database. The user interface is similar to the desktop version shown in Figure 17.1.

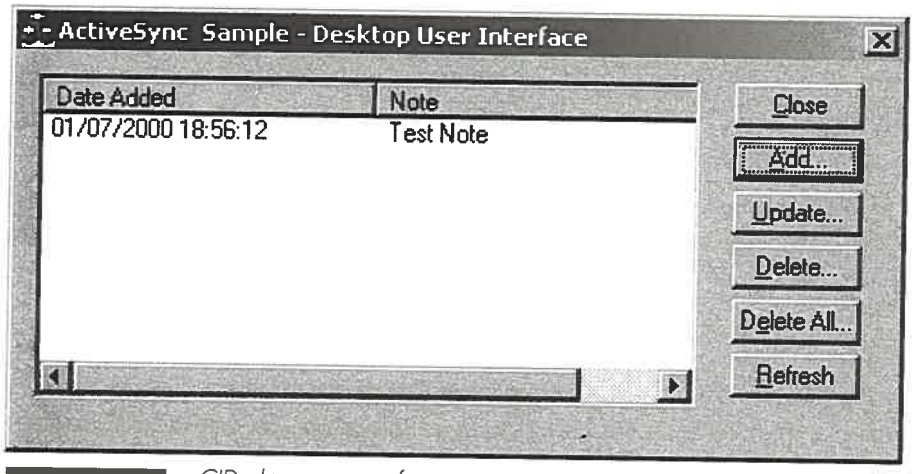


Figure 17.1 *CIDesktop user interface*

2. **ASDevice**—A standard Windows CE DLL that implements the Windows CE side of the ActiveSync provider, located in `\ActiveSync\ASDevice`.
3. **CIDesktop**—A desktop PC MFC application, located in `\ActiveSync\ASDesktop\CLDesktop`, that presents a simple user interface to manipulate the records stored in a flat file. Once again, the code is straightforward and not described here. The user interface is illustrated in Figure 17.1.
4. **ASDesktop**—A desktop PC DLL that implements a COM component with the `IReplObjHandler` and `IReplStore` interfaces.

The code in ASDevice and ASDesktop is organized to isolate the application-specific data access code, and so can be used as a skeleton for implementing your own ActiveSync provider.

Installation and Registration

Installing an ActiveSync provider on a Windows CE device requires the following steps:

1. Copy the DLL (for example, ASDevice.dll) into a suitable directory, such as "`\Windows`."
2. Add an entry such as the following in the Synchronization registry key for your ActiveSync provider:

```
HKEY_LOCAL_MACHINE
  Windows CE Services
    Synchronization
      Objects
        Appointment
        Contact
        Tasks
        AsyncSample
```

3. Add a `"Store" REG_SZ` value to this key that contains the name of the `.DLL` that implements the ActiveSync provider. This should contain the fully qualified path if the `.DLL` is not in a standard location (such as the root or `\Windows` directory).

```
AsyncSample
  Store ASDevice.dll
```

Code to register the DLL is contained in an exported function called `RegisterActiveSync` in `ASDevice.CPP`. This function is called from the CLDevice application when the "Register" button is pressed.

4. Copy the user interface application (for example, `CLDevice.exe`) into a suitable directory, such as the root, and run the application. This creates

the database (`ActiveSyncNotes`) that will contain the synchronized items. With the sample application you should click the "Register" button to add the necessary registry items.

Installing the ActiveSync provider on the desktop PC requires more work, since a COM component is being registered. Here are the steps:

1. Copy the application (`CLDesktop.exe`) into any suitable directory. Running this application will create the file "`\ActiveSynNotes.dat`" used to store the items. The user interface is almost identical to `CLDevice.exe` except that the "Register" button is replaced by "Refresh." Note that the list of items is not automatically updated, so you will need to click "Refresh" to ensure that the list is up to date.
2. Copy the DLL (for example, `ASDesktop.dll`) into any suitable directory. You will need to register the COM component using the `REGSVR32` application:

```
REGSVR32 ASDesktop.dll
```

As well as writing the standard registry entries for a COM component, entries specific to an ActiveSync provider are added. The code to add COM component entries is contained in the function `DllRegisterServer` in `COMDLL.CPP`. This calls the function `RegisterActiveSync` in `COMDLL.CPP` to add the ActiveSync provider registry entries.

A new key with the same name used on the Windows CE device (for example, "`AsyncSample`") is added in the following location:

```
HKEY_LOCAL_MACHINE
  Software
    Microsoft
      Windows CE Services
        Services
          Synchronization
            Objects
              Appointment
              Contact
              Task
              AsyncSample
```

It is important that the Windows CE device and desktop PC use the same key names, since this forms the link between the two sides of the ActiveSync provider. On the desktop PC the key contains the following values:

```
AsyncSample
  [Default]          "ActiveSync Example Provider"
  Display Name       "TestNote"
  Plural Name        "TestNotes"
  Store              "Asdesktop.ActiveSyncEg"
  Disabled           0
```

The `"[default]"`, `"Display Name"`, and `"Plural Name"` `REG_SZ` string entries are used by ActiveSync to display information about the provider's status. The `"Store"` `REG_SZ` string contains the ProgID of the desktop COM Component that implements the ActiveSync provider. This string is the same value used when the DLL (for example, `ASDesktop.DLL`) registers its COM component. ActiveSync uses this value to locate the COM component and uses the COM registry entry `"InProcServer"` to find the fully qualified pathname for the DLL's location. The `"Disabled"` value (a `REG_DWORD`) has a value of 0 if the provider is active, or 1 if it is temporarily disabled.

AsyncSample is actually a folder, or object type (the terms mean the same). A store can implement multiple folders by having several object types (for example, Appointment, Contact, and Task) with the same store.

The desktop ActiveSync registry settings are copied into each desktop PC profile under the `HKEY_CURRENT_USER` key, using the same key names as described above. Now, when you run the ActiveSync user interface, you will see a new entry for this provider (Figure 17.2).

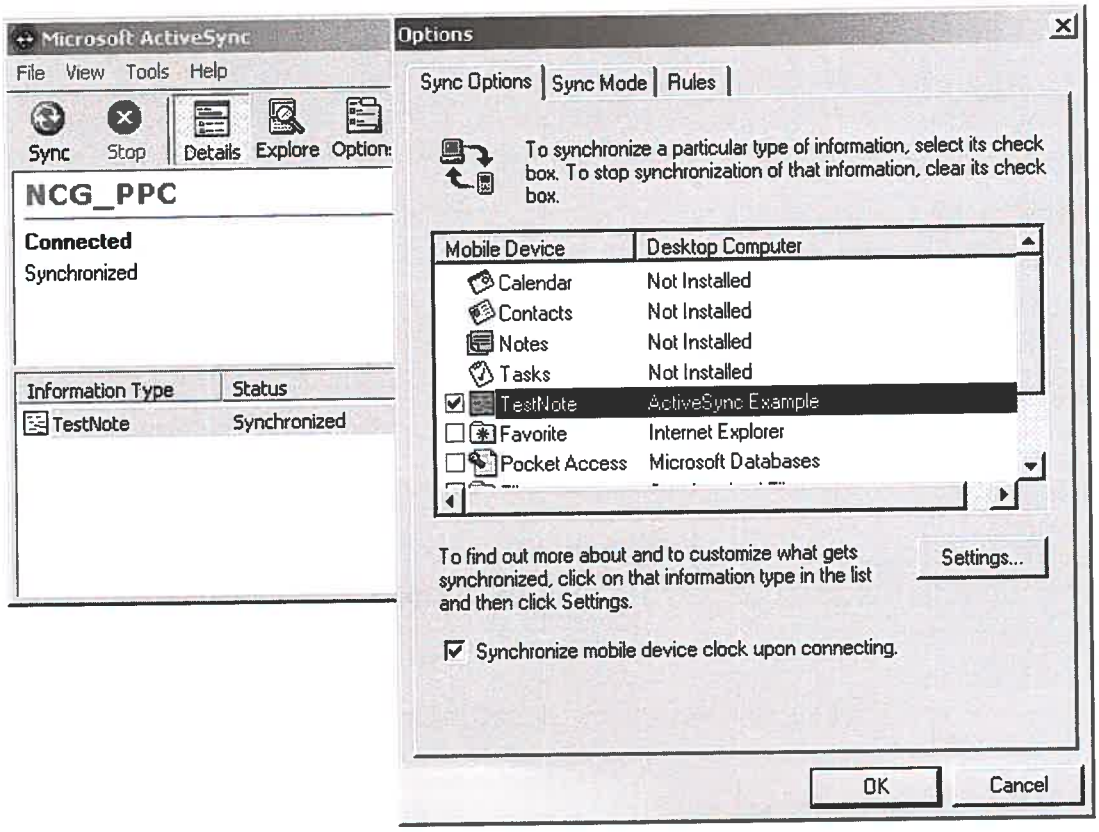


Figure 17.2 *ActiveSync with an additional service*

Data Organization

Each item in the database consists of the following:

- The timestamp of when the item was created, using a `FILETIME` structure
- The note itself, up to 256 Unicode characters

The timestamp is the field's unique identifier and is never changed. The user interfaces allow notes to be added, deleted, or updated, and the ActiveSync provider synchronizes these items.

Important Note

This ActiveSync example implements a simple provider. The description of the function arguments, structure members, and interface functions only includes those elements required to implement a fully functional yet simple provider. You should refer to the Windows CE documentation for full descriptions of all functions, structure members, and so on.

Implementing the Windows CE Device Provider

ActiveSync on a Windows CE device is based around store objects such as files, directories, databases, and database records. As described in Chapter 4, each object has a unique object identifier or `CEOID`. Any object that has a `CEOID` can be an item synchronized by ActiveSync. Thus, you can synchronize files, directories, databases, or database records. You can use these objects to hold more than one item (for example a file might contain many records, each of which is an item), but you will need to manage lists of these items, and this gets more complex. The simplest approach is to represent an ActiveSync item by a single database record.

Windows CE 2.1 and later versions allow databases to be created in volumes that can be located on, for example, storage cards. Additional functions (`FindObjects` and `SyncData`) are provided to synchronize databases in these volumes. Note that some storage cards and other media do not use the `CEOID` object identifiers for the file system, so they are more difficult to synchronize.

Remember, the Windows CE ActiveSync DLL is not a COM component. However, the DLL does implement the interface `IReplObjHandler`. The DLL itself is responsible for creating an instance of this interface (usually through implementing the interface using a C++ class), and not any external application. Therefore, all the usual COM elements, such as class factories, `DLLGetObject`, and other exported functions, are not required.

First, let's look at the functions the ActiveSync provider must export.

InitObjType Exported Function

InitObjType, located in ASDevice.cpp, is called by ActiveSync when the provider is started and terminated. This occurs when the Windows CE device connects or disconnects. This function carries out any initialization/termination required by the provider and returns a pointer to the IReplObjHandler interface.

Listing 17.1 *Implementation of InitObjType*

```
extern "C" BOOL _declspec(dllexport) InitObjType(
      LPWSTR lpszObjType,
      IReplObjHandler **ppObjHandler,
      UINT uPartnerBit)
{
  if ( lpszObjType == NULL )
  {
    // Terminates the device provider module and
    // frees all allocated resources.
    return TRUE;
  }
  // Allocate a new IReplObjHandler.
  *ppObjHandler = new CDataHandler;
  // Save the uPartnerBit so that you can use it later on
  g_uPartnerBit = uPartnerBit;
  // Find Object Identifier of our database
  g_oidDataBase = ASGetDBOID(DB_NAME);
  return TRUE;
}
```

In Listing 17.1 the C++ class CDataHandler implements the IReplObjHandler interface, so an instance of the class is created and a pointer returned through the parameter ppObjHandler.

A Windows CE device can maintain synchronization with up to two desktop PCs. The uPartnerBit has a value 1 when synchronizing with the first partnership and 2 with the second. Maintaining two partnerships is more complex; you can choose to support only one partnership, as is the case with the EMail ActiveSync option. In this code the partnership bit is saved in the global variable g_uPartnerBit.

Lastly, the CEOID of our database is stored in the global variable g_oidDatabase. The function ASGetDBOID is located in DB.CPP, together with the other database access code for the provider.

ObjectNotify Exported Function

Windows CE constantly monitors changes in the object store. When a change occurs (for example, a database record is updated), all loaded ActiveSync providers are notified of the change through a call to ObjectNotify. This

function determines the nature of the change (whether it was a file, directory, database, or record change) and returns TRUE if it is an item this provider can synchronize.

The provider should ensure that the record is in its own database. There is no point synchronizing someone else's database! This is done through the function ASRecInDB to be found in DB.CPP, which determines the parent CEOID for the database record by calling CEOidGetInfo, and checks that this is the same as the CEOID for our database.

Listing 17.2 *Implementation of ObjectNotify*

```
extern "C" BOOL _declspec(dllexport) ObjectNotify(
      POBJNOTIFY pNotify)
{
  // Check to see if the structure size
  // is the smaller (version control).
  if ( pNotify->cbStruct < sizeof( OBJNOTIFY ) )
  {
    MessageBox(NULL,
      _T("ObjectNotify—incorrect version"),
      NULL, MB_OK);
    return FALSE;
  }
  // We're only interested in database record
  // changes or clear change notifications
  if(!(pNotify->uFlags & (ONF_RECORD |
        ONF_CLEAR_CHANGE)))
    return FALSE;
  // For non-deleted records, check that the
  // record is in our database
  if(!(pNotify->uFlags & ONF_DELETED))
  {
    if(!(pNotify->uFlags & ONF_RECORD))
      // it's not actually a record, so ignore
      return FALSE;
    if(!ASRecInDB(g_oidDataBase, pNotify->oidObject))
      // not in our database
      return FALSE;
  }
  // sets the oid of the object to be replicated
  pNotify->poid = (UINT*) &pNotify->oidObject;
  // if object is to be deleted, set the
  // number of objects to be deleted
  if(pNotify->uFlags & ONF_DELETED)
    pNotify->cOidDel = 1;
  else
    pNotify->cOidChg = 1;
  return TRUE;
}
```

ObjectNotify is passed a pointer to an OBJNOTIFY structure. The members of this structure used in this sample are shown in Table 17.1.

Table 17.1 *OBJNOTIFY structure members*

Member	Description
cbStruct	The size of the structure being passed in. The provider should check this against the size of structure it is using to ensure version compatibility. This should be done for all structures passed to provider functions.
uFlags	Contains flags indicating the type of change. For example, ONF_RECORD indicates a database record has changed, ONF_CLEAR_CHANGE indicates that the change bit for the object should be cleared, and ONF_DELETED indicates that a record has been deleted.
oidObject	CEOID of the item being notified.
poid	Set by the provider to be a pointer to the CEOID of the item to be synchronized. In simple providers, this will generally be the CEOID of the item passed into ObjectNotify through the member oidObject. More complex providers can set an array of CEOIDs to be synchronized.
cOidDel	Number of items to be deleted. This will be one if uFlags is set to ONF_DELETED for simple providers.
cOidChg	Number of items to be changed. This will be one if uFlags is set to ONF_RECORD for simple providers.

GetObjTypeInfo Exported Function

ActiveSync on the Windows CE device calls this exported function when it needs information about the database being synchronized. The function fills in members of the OBJTYPEINFO structure, such as the following:

- **szName**—The name of the database.
- **cObjects**—The number of items to be synchronized (which for simple providers is the number of database records).
- **cbAllobj**—The overall size of the items to be synchronized. This is equal to the size of the database in bytes.
- **ftLastModified**—A FILETIME structure containing the time and date of when the database was last changed.

Listing 17.3 shows the implementation of GetObjTypeInfo from the file ASDevice.CPP.

Listing 17.3 *Implementation of GetObjTypeInfo*

```
extern "C" BOOL _declspec(dllexport) GetObjTypeInfo
      (POBJTYPEINFO pInfo)
{
  CEOIDINFO oidInfo;
  // Check versioning of the structure
  if ( pInfo->cbStruct < sizeof( OBJTYPEINFO ) )
  {
    MessageBox(NULL,
      _T("GetObjTypeInfo called-wrong version"),
      NULL, MB_OK);
    return FALSE;
  }
  // Clear the structure.
  memset( &(oidInfo), 0, sizeof(oidInfo));
  // Retrieves information about the object
  // in the object store.
  CeOidGetInfo( g_oidDataBase, &oidInfo );
  // Store the database information into
  // the OBJTYPEINFO structure.
  wcscpy( pInfo->szName,
    oidInfo.infDatabase.szDbaseName );
  pInfo->cObjects = oidInfo.infDatabase.wNumRecords;
  pInfo->cbAllObj = oidInfo.infDatabase.dwSize;
  pInfo->ftLastModified =
    oidInfo.infDatabase.ftLastModified;
  return TRUE;
}
```

Note that the version of the `OBJTYPEINFO` structure is checked. Information about the database is obtained through a call to the Windows CE function `CeOidGetInfo`, which fills in a `CEOIDINFO` structure.

Implementing the Device IReplObjHandler COM Interface

`IReplObjHandler` functions are responsible for converting your items (such as database records) into a stream of bytes (serialization) or converting a stream of bytes into an item (deserialization). Serialization and deserialization are required so that items can be transferred between Windows CE devices and desktop PCs.

The desktop PC ActiveSync provider also needs to implement the `IReplObjHandler` and should serialize and deserialize items using the same data format. However, the implementations are typically different (since the item stores are not the same), so it is usually best to keep to separate code implementations.

In the example, `IReplObjHandler` is implemented by the C++ class `CDataHandler` that is declared in `ReplObjHandler.h` and implemented in

ReplObjHandler.cpp. Since IReplObjHandler is declared as a COM interface, IUKnown must be implemented. However, since we are only implementing a COM interface and not an entire COM component, these implementations are very straightforward. AddRef and Release simply increment and decrement a reference count. QueryInterfce always returns E_NOINTERFACE—this function will never actually be called.

An overview of the essential IReplObjHandler interface functions is provided in Table 17.2.

Table 17.2 *IReplObjHandler interface functions*

Function	Description
Setup	Called when serialization or deserialization of an item is about to begin.
Reset	Called when serialization or deserialization is completed.
GetPacket	Serialization. This function is called to convert an item into a stream of bytes, which occurs when an item is being sent from the Windows CE device to desktop PC. Large items need to be split up into packets, and GetPacket will be called once for each packet. Small items can be serialized in a single packet. The function returns NOERROR if more packets are required, or RWRN_LAST_PACKET if this is the last or only packet.
SetPacket	Deserialization. This function is called when an item needs to be converted from a stream of bytes to an item. This occurs when an item is being sent from the desktop PC to Windows CE device. Large items are divided into packets, and a call to SetPacket is made for each packet. The item is written out to the database when all packets are received. This could result in an existing record being updated, or a new record added.
DeleteObject	Called when ActiveSync detects that an item must be deleted from the Windows CE device database.

Serialization Format

You will need to determine the format to be used for serialization and deserialization. GetPacket and SetPacket provide LPBYTE pointers, but this can be cast to any pointer you like. In the example, a typedef for a structure called NOTE is used, and this contains the creation FILETIME (the unique identifier), the last modify timestamp as a FILETIME, and a Unicode string (Listing 17.4). It is declared in db.h.

Listing 17.4 *Structure NOTE*

```
typedef struct tagNOTE
{
  FILETIME ftOriginal;      // time when note was created
  FILETIME ftLastUpdate;    // time last updated
  WCHAR szNote[STRLEN_NOTE];
} NOTE;
```

Note that the last modify timestamp member is not used on the device, and so is not strictly required in this structure. However, this same structure is used on the desktop PC for storing data, so it is convenient to leave it here.

It is essential that exactly the same format is used by the ActiveSync provider on the Windows CE device and the desktop PC. Note how the Unicode string has been declared as `WCHAR` and not `TCHAR`. This ensures that it is defined as a Unicode string even if the code is compiled for ANSI (which is most often the case on the desktop PC).

IReplObjHandler::Setup

This function is called by ActiveSync before any item is received or sent. This provides an opportunity to perform any initialization. A pointer to a `REPLSETUP` structure is passed in, and this provides information such as the direction of the transfer.

The Setup function will normally save a pointer to the `REPLSETUP` structure for future use. Since ActiveSync is multithreaded, it is possible that a read (outgoing transfer) occurs at the same time as a write (incoming transfer). Therefore, the `IReplObjHandler` class has two members, `m_pReadSetup` and `m_pWriteSetup`, to store separate pointers (Listing 17.5). The pointer will be used later in `GetPacket` and `SetPacket`.

Listing 17.5 *IReplObjHandler::Setup implementation*

```
STDMETHODIMP CDataHandler::Setup(PREPLSETUP pSetup)
{
  // Can be reading and writing at the same time, so need
  // two setups
  if(pSetup->fRead)
    m_pReadSetup = pSetup;
  else
    m_pWriteSetup = pSetup;
  return NOERROR;
}
```

Most `REPLSETUP` members are only used on a desktop implementation of `IReplObjHandler`. Those listed below may be used on the Windows CE device.

- **`fRead`**—`TRUE` if Setup is being called for reading an item, `FALSE` for a write
- **`Oid`**—`CEOID` of the item, for example, an existing record in the database that is being sent to the desktop
- **`oidNew`**—Set to the `CEOID` of the item that has been added or updated in the CE property database
- **`dwFlags`**—Contains the value `RSF_NEW_OBJECT` if this is a new record to be added to the CE property database

IReplObjHandler::Reset

The `Reset` function provides an opportunity to free any resources created during serialization or deserialization. In this case, there is nothing to do (Listing 17.6).

Listing 17.6 *IReplObjHandler::Reset implementation*

```
STDMETHODIMP CDataHandler::Reset(PREPLSETUP pSetup)
{
  return NOERROR;     // no resources to be freed
}
```

IReplObjHandler::GetPacket

ActiveSync calls this function to request a packet for a particular item being synchronized. Your implementation should produce a byte stream representing the entire item (if it fits into a single packet) or the next packet in sequence. The function passes in the recommended maximum size of the packet in `cbRecommend`.

Listing 17.7 shows the implementation of `GetPacket`. The function calls `ASSerializeRecord` (located in `db.cpp`) to read the record for the given `CEOID` (`m_pReadSetup->oid`). The function serializes the record into a `NOTE` structure, returns a pointer to the structure in `lpByte`, and returns the size of the `NOTE` structure in `dwLen`. The pointer and size are returned to ActiveSync through the parameters `lppbData` and `pcbData`.

Listing 17.7 *IReplObjHandler:: GetPacket implementation*

```
STDMETHODIMP CDataHandler::GetPacket(LPBYTE *lppbData,
    DWORD *pcbData, DWORD cbRecommend)
{
  HRESULT hr = RWRN_LAST_PACKET;
  LPBYTE lpByte;
  DWORD dwLen;

  if(!ASSerialiseRecord(m_pReadSetup->oid,
        &lpByte, &dwLen))
    hr = RERR_BAD_OBJECT;
  else
  {
    *lppbData = lpByte;
    *pcbData = dwLen;
  }
  return hr;
}
```

GetPacket returns RWRN_LAST_PACKET if the serialization was successful and this is the last or only packet. RERR_BAD_OBJECT is returned if the object could not be serialized.

IReplObjHandler::SetPacket

SetPacket does the opposite of GetPacket—it is passed a pointer to a stream of bytes and writes the data to a new or existing record in the database. The REPLSETUP structure member dwFlags contains the value RSF_NEW_OBJECT if this item is a new record; otherwise, an existing record is to be updated.

In Listing 17.8, SetPacket casts the incoming lpbData pointer to a NOTE pointer and calls ASDeserializeRecord (located in db.cpp) to perform the update. The REPLSETUP structure member oid contains the CEOID of the record to be updated, or 0 if this is a new record.

Listing 17.8 *IReplObjHandler:: SetPacket implementation*

```
STDMETHODIMP CDataHandler::SetPacket(LPBYTE lpbData,
        DWORD cbData)
{
  NOTE* aNote;

  CEOID oidNewRec;
  BOOL bNewRec;

  aNote = (NOTE*)lpbData;

  bNewRec = m_pWriteSetup->dwFlags & RSF_NEW_OBJECT;

  if((oidNewRec = ASDeserializeRecord(&aNote->ftOriginal,
        &aNote->ftLastUpdate,
        aNote->szNote,
        wcslen(aNote->szNote),
        bNewRec, m_pWriteSetup->oid)) == 0)
    return RERR_SKIP_ALL;
  else
  {
    m_pWriteSetup->oidNew = oidNewRec;
    return NOERROR;
  }
}
```

SetPacket returns RERR_SKIP_ALL if the update fails. This will cause all subsequent packets to be discarded. If successful, the CEOID of the new record is assigned to the oidNew member of REPLSETUP, and the function returns NOERROR.

IReplObjHandler::DeleteObj

`DeleteObj` is called when an item needs to be deleted from the database. The function is passed a `REPLSETUP` pointer as a parameter and calls `ASDeleteRecord` (located in `DB.CPP`) to delete the record (Listing 17.9).

Listing 17.9 *IReplObjHandler:: DeleteObj implementation*

```
STDMETHODIMP CDataHandler::DeleteObj(PREPLSETUP pSetup)
{
  if(ASDeleteRecord(pSetup->oid))
    return NOERROR;
  else
    return E_UNEXPECTED;
}
```

Implementing the Desktop Provider

Implementing the desktop provider takes more time and effort. This is, in the main, because ActiveSync makes no assumptions about where items are stored and when they are changed.

You need to create a full COM component with a class factory, registration, and other features. The component will need to implement the COM component `IReplStore`, which is used to manage the store, folder, and item manipulation. The desktop provider also needs to implement the `IReplObjHandler` interface.

Representing HREPLITEM and HREPLFLD

`HREPLITEM` and `HREPLFLD` are pointers used by ActiveSync to point at your data associated with items and folders. In certain circumstances, ActiveSync passes a `HREPLOBJ` that can point either to a `HREPLITEM` or `HREPLFLD`. You therefore need a storage mechanism that can store either a `HREPLITEM` or `HREPLFLD` and be able to determine which type is currently being stored. Perhaps the most straightforward technique is to use a structure containing a union (Listing 17.10).

Listing 17.10 *REPLOBJECT structure*

```
// structure and define for HREPLITEM and
// HREPLFLD structures
#define RT_ITEM     1
#define RT_FOLDER   2
```

```
typedef struct tagREPLOBJECT
{
  // uType indicates if a folder (RT_FOLDER)
  // or item (RT_ITEM) is currently being stored
  UINT uType;
  // Create a union so folder and item information can be
  // stored in the same structure
  union
  {
    // for folder, has the contents been changed
    BOOL fChanged;
    // for item, creation time (unique identifier)
    // and last modify time
    struct
    {
      FILETIME ftCreated, ftModified;
    };
  };
} REPLOBJECT, *LPREPLOBJECT;
```

The member `uType` can contain either `RT_ITEM` or `RT_FOLDER`; they indicate the current use of the structure. Folders use the member `fChanged` and items use `ftCreated` or `ftModified`.

ActiveSync uses the data you place in this structure to track changes to folders and items. There is generally a separate structure for each item and folder in store, and ActiveSync stores these structures in `repl.dat`.

You can place any type of data in the structure that is applicable to your application. However, you should attempt to limit the amount of data you store in the structure. There will always be far more items than folders, so you should focus on the amount of data stored in the item. If you use a structure with a union (as shown in Listing 17.10), ensure that the amount of data associated with the folder is less than that used for the item—the overall size of the structure is determined by the largest members in the union.

Storing Data on the Desktop

ActiveSync makes no assumptions about where the data being synchronized is being stored—you can use flat files, local databases, or server databases. In this example a simple flat file is used, and each item is stored as a `NOTE` structure (which has a fixed size). The code to access this file is located in `ListDB.h`, and uses the standard file I/O techniques outlined in Chapter 2.

Implementing IReplStore

The `IReplStore` COM interface declares functions that ActiveSync uses to obtain information and manipulate the store, folder, and items being synchronized. Table 17.3 shows the functions categorized by function.

Table 17.3 *IReplStore interface functions*

Category	Functions
Initialization	`Initialize`
Store information and manipulation	`GetStoreInfoCompareStoreIDs`
Folder information and manipulation	`GetFolderInfoIsFolderChanged`
Iterate all items in a folder	`FindFirstItemFind` `NextItemFindItemClose`

Manipulate `HREPLITEM` or `HREPLFLD` objects	`ObjectToBytes`
	`BytesToObject`
	`FreeObjectCopy`
	`ObjectIsValidObject`

`HREPLITEM` item synchronization	`CompareItem`
	`IsItemChanged`
	`IsItemReplicated`
	`UpdateItem`

Configuration dialog, provider icon, and name information and activity reporting	`ActivateDialog` `GetObjTypeUIDataReportStatus`
Conflict resolution and duplicate removal	`GetConflictInfoRemoveDuplicates`

Because `IReplStore` is a COM interface, you must provide an implementation of all these functions; otherwise, your provider will not compile. However, only the functions shown in this chapter are essential in a simple provider.

In the example, `IReplStore` is implemented by the class `CActiveSyncEg`. The declaration of this class is in `Component.h`, and the implementation of `IReplStore` is in the file `IReplStore.cpp`.

IReplStore Initialization

ActiveSync calls the `IReplStore::Initialize` function when the provider is first loaded (Listing 17.11). The function passes a pointer to a `IReplNotify` interface provided by ActiveSync and used by the provider to notify ActiveSync when item changes occur in the store. Since the example doesn't implement continuous synchronization, this pointer is ignored. The `uFlags` parameter will contain `ISF_REMOTE_CONNECTED` if synchronization is being carried out over a dialup or other type of remote connection. In this case, you should avoid anything that requires user intervention (such as showing a dialog).

Listing 17.11 *IReplStore::Initialize implementation*

```
STDMETHODIMP CActiveSyncEg::Initialize(
      IReplNotify*pNotify,UINT uFlags )
```

```
{
  m_bInitialized = TRUE;
  return NOERROR;
}
```

Some IReplStore functions can be called before Initialize is called, so you should be careful not to rely on this initialization. The functions are GetStoreInfo, GetObjTypeUIDate, GetFolderInfo, ActivateDialog, BytesToObject, ObjectToBytes, and ReportStatus.

Store Information and Manipulation

ActiveSync calls the function GetStoreInfo (Listing 17.12) and passes a pointer to a STOREINFO structure that the provider populates with information about its store.

Listing 17.12 *IReplStore:: GetStoreInfo implementation*

```
STDMETHODIMP CActiveSyncEg::GetStoreInfo(
      PSTOREINFO pStoreInfo )
{
  // Check correct version of StoreInfo structure
  if(pStoreInfo->cbStruct < sizeof(*pStoreInfo))
  {
    MessageBox(NULL,
      _T("GetStoreInfo—Invalid Arg"), NULL, 0);
    return E_INVALIDARG;
  }
  // we only support single-threaded operation
  pStoreInfo->uFlags = SCF_SINGLE_THREAD;
  // Set store's progid and description
  strcpy(pStoreInfo->szProgId, g_szVerIndProgID);
  strcpy(pStoreInfo->szStoreDesc, g_szFriendlyName);
  // this is as far as we get if we're not Initialized
  if(!m_bInitialized)
  {
    return NOERROR;
  }
  // Create the store's unique identifier—
  // Set the length of the store identifier
  pStoreInfo->cbStoreId =
      (strlen(g_szStoreFile) + 1) * sizeof(TCHAR);
  // ActiveSync calls GetStoreInfo twice. Once to
  // get the size of the store id (when lpbStoreId is
  // NULL), and a second time, providing a buffer pointed
  // to by lpbStoreId where the store id can be placed.
  if(pStoreInfo->lpbStoreId == NULL)
    return NOERROR;
```

```
  memcpy(pStoreInfo->lpbStoreId, g_szStoreFile,
    (strlen(g_szStoreFile) + 1) * sizeof(TCHAR));
  return NOERROR;
}
```

The function `GetStoreInfo` is called twice by ActiveSync, the first time to determine the size of the buffer required to hold the store's unique id (`cbStoreId`), and the second time to copy the store id into a buffer (`lpbStoreId`). Table 17.4 describes the `STOREINFO` members used by this implementation of `GetStoreInfo`.

Table 17.4 *StoreInfo members used in CActiveSyncEg::GetStoreInfo*

Member	Purpose
`uFlags`	Use `SCF_SINGLE_THREAD` if your provider is single-threaded.
`szProgId`	The store's ProgID, such as "`Asdesktop.ActiveSyncEg`".
`szStoreDesc`	Description of store displayed to user, such as "`ActiveSync Example`".
`cbStoreId`	Length of the store's id in bytes.
`lpbStoreId`	Store's unique id, for example, the name of the data file "`\ActiveSynNotes.dat`".

The function `CompareStoreIDs` is called by ActiveSync to determine if two store ids are actually the same. Listing 17.13 compares `cbID1` and `cbID2` to determine whether the number of bytes in the store ids are the same and, if they are, uses `memcmp` to perform a byte-wise comparison of the two strings. The function returns 0 if the `lpbID1` and `lpbID2` are ids that refer to the same store.

Listing 17.13 *IReplStore::CompareStoreIDs implementation*

```
STDMETHODIMP_(int) CActiveSyncEg::CompareStoreIDs
      (LPBYTE lpbID1, UINT cbID1,
      LPBYTE lpbID2, UINT cbID2)
{
  if(cbID1 < cbID2)
    // first store is smaller than the second store
    return -1;
  if(cbID1 > cbID2)
    // first store is larger than the second store
    return 1;
  // now compare the store ids byte by byte.
  return memcmp(lpbID1, lpbID2, cbID1);
}
```

Folder Information and Manipulation

A store can contain one or more folders in which items are placed. ActiveSync calls `GetFolderInfo` for each object type (for example, "`AsyncSample`") configured in the registry for the provider. The implementation of `GetFolderInfo` returns a pointer to the `IReplObjHandler` interface associated with this folder (the `m_DataHandler` member is a `CDataHandler` class object that implements `IReplObjHandler`) and to a `HREPLFLD` object (Listing 17.14).

Listing 17.14 *IReplStore:: GetFolderInfo implementation*

```
STDMETHODIMP CActiveSyncEg::GetFolderInfo(LPSTR
    lpszObjType,
    HREPLFLD *phFld, IUnknown ** ppObjHandler)
{
  LPREPLOBJECT pFolder = (LPREPLOBJECT) *phFld;
  if(pFolder == NULL)            // new folder required
  {
    pFolder = new REPLOBJECT;
  }
  pFolder->uType = RT_FOLDER;
  pFolder->fChanged = TRUE;
  *phFld = (HREPLFLD)pFolder;
  // CDataHandler member m_DataHandler
  // implements IReplObjHandler
  *ppObjHandler = &m_DataHandler;
  return NOERROR;
}
```

In the example, `HREPLFLD` is actually a pointer to a `REPLOBJECT`. The `HREPLFLD` parameter can be `NULL`, in which case a new `REPLOBJECT` is created, or, if not `NULL`, the existing `REPLOBJECT` is used.

The function `IsFolderChanged` is called by ActiveSync to determine whether items in a folder need to be synchronized. With continuous synchronization this function is called frequently, but with manual synchronization it is only called when synchronization starts. In Listing 17.15 the function always sets `pfChanged` to `TRUE`, indicating that the folder needs to be synchronized.

Listing 17.15 *IReplStore:: IsFolderChanged implementation*

```
STDMETHODIMP CActiveSyncEg::IsFolderChanged(HREPLFLD hFld,
      BOOL *pfChanged )
{
  *pfChanged = TRUE;
  return NOERROR;
}
```

Iterate Items in a Folder

ActiveSync requests the provider to iterate through the items in a folder by calling the functions `FindFirstItem`, `FindNextItem`, and `FindItemClose`. The provider reads the first item (`FindFirstItem`) or the next item (`FindNextItem`) from the store and creates a `HREPLITEM` for each item. In the example, the functions `GetFirstNote` or `GetNextNote` (located in `db.cpp`) read the items, and a `HREPLITEM` is created represented by the `REPLOBJECT` structure (Listing 17.16). The three `REPLOBJECT` members are initialized with appropriate values read from the store. The `HREPLITEM` item is returned through the `phItem` parameter.

Listing 17.16 *IReplStore:: FindFirstItem and FindNextItem implementations*

```
// Returns an HREPLITEM structure for the first item in
// the .DAT file. The data in HREPITEM is the OriginalTime
// (the unique identifier) and ModifyTime
// (to determine if the item has changed);

STDMETHODIMP CActiveSyncEg:: FindFirstItem(HREPLFLD hFld,
      HREPLITEM *phItem, BOOL *pfExist )
{
  WCHAR szNote[STRLEN_NOTE];
  FILETIME ftCreateTime, ftModifyTime;
  // attempt to get first record
  *pfExist = m_ListDB.GetFirstNote(&ftCreateTime,
      szNote, &ftModifyTime);
  if(!*pfExist)
    return NOERROR;
  // now make up the HREPLFLD
  LPREPLOBJECT lpRepl = new REPLOBJECT;
  lpRepl->uType = RT_ITEM;
  lpRepl->ftCreated = ftCreateTime;
  lpRepl->ftModified = ftModifyTime;
  // set our pointer into HREPLITEM
  *phItem = (HREPLITEM)lpRepl;
  return NOERROR;
}

// Find the next item from the .DAT file
STDMETHODIMP CActiveSyncEg::FindNextItem(HREPLFLD hFld,
      HREPLITEM *phItem, BOOL *pfExist )
{
  WCHAR szNote[STRLEN_NOTE];
  FILETIME ftCreateTime, ftModifyTime;

  // attempt to get first record
  *pfExist = m_ListDB.GetNextNote(&ftCreateTime,
      szNote, &ftModifyTime);
```

```
  if(!*pfExist)
  {
    return NOERROR;
  }
  // now make up the HREPLFLD
  LPREPLOBJECT lpRepl = new REPLOBJECT;
  lpRepl->uType = RT_ITEM;
  lpRepl->ftCreated = ftCreateTime;
  lpRepl->ftModified = ftModifyTime;
  // set our pointer into HREPLITEM
  *phItem = (HREPLITEM)lpRepl;
  return NOERROR;
}
```

`FindFirstItem` and `FindNextItem` set the `pfExist BOOL` parameter to `TRUE` if a `HREPLITEM` is returned, or `FALSE` if no more items exist. `FindItemClose` is called when `pfExist` is set to `FALSE` (Listing 17.17).

Listing 17.17 *IReplStore:: FindItemClose implementation*

```
// Finished going through all records.
// Nothing to do in this case.

STDMETHODIMP CActiveSyncEg::FindItemClose(HREPLFLD hFld )
{
  return NOERROR;
}
```

Manipulating HREPLITEM and HREPLFLD Objects

The provider must implement functions that allow ActiveSync to manipulate `HREPLITEM` and `HREPLFLD` objects. Table 17.5 shows the functions that must be implemented. Many of these functions operate on `HREPLITEM` or `HREPLFLD` objects, and the functions may need to take different actions depending on which is passed. Remember that the generic type `HREPLOBJ` is used to refer to both `HREPLITEM` and `HREPLFLD` objects.

Table 17.5 *Functions to manipulate HREPLITEM and HREPLFLD objects*

Function	Description
`ObjectToBytes`	Convert a `HREPLITEM` or `HREPLFLD` into a stream of bytes.
`BytesToObject`	Convert a stream of bytes into a `HREPLITEM` or `HREPLFLD`.
`FreeObject`	Free memory used by a `HREPLITEM` or `HREPLFLD`.
`CopyObject`	Copy one `HREPLITEM` or `HREPLFLD` into another.
`IsValidObject`	Determine whether a `HREPLITEM` or `HREPLFLD` still represents a valid object.

Using the structure/union REPLOBJECT to store both HREPLITEM and HREPLFLD objects greatly simplifies the coding of these functions.

ActiveSync calls ObjectToBytes and BytesToObject when writing and reading objects to and from the file repl.dat. ObjectToBytes (Listing 17.18) will be called twice for each conversion. In the first call, ObjectToBytes simply returns the number of bytes required to write the object. In the second call ActiveSync provides a buffer of the correct length into which the copy is made.

Listing 17.18 *IReplStore:: ObjectToBytes implementation*

```
STDMETHODIMP_(UINT) CActiveSyncEg::ObjectToBytes
      (HREPLOBJ hObject, LPBYTE lpb )
{
  // buffer has been created to requested size
  if(lpb != NULL)
    memcpy(lpb, (LPREPLOBJECT)hObject,
        sizeof(REPLOBJECT));
  return sizeof(REPLOBJECT);
}
```

Both HREPLITEM and HREPLFLD objects are passed into ObjectToBytes, but because REPLOBJECT is used for both folders and items, the same code can be used for both.

BytesToObject (Listing 17.19) creates a new REPLOBJECT, copies from the stream of bytes into this new structure, and returns a pointer cast to a HREPLOBJ.

Listing 17.19 *IReplStore:: BytesToObject implementation*

```
STDMETHODIMP_(HREPLOBJ) CActiveSyncEg::BytesToObject
        (LPBYTE lpb, UINT cb )
{
  if(cb != sizeof(REPLOBJECT))
    MessageBox(NULL,
      _T("Not correct size in Bytes to object"),
      NULL,0);
  LPREPLOBJECT lpReplObject = new REPLOBJECT;
  // perform the copy
  memcpy(lpReplObject, lpb, cb);
  return (HREPLOBJ)lpReplObject;
}
```

ActiveSync calls FreeObject (Listing 17.20); the implementation should free any memory associated with the HREPLOBJ object.

Listing 17.20 *IReplStore:: FreeObject implementation*

```
STDMETHODIMP_(void) CActiveSyncEg::FreeObject(
      HREPLOBJ hObject )
{
  LPREPLOBJECT pItem = (LPREPLOBJECT)hObject;
  delete (LPREPLOBJECT) hObject;
}
```

From time to time, ActiveSync needs to copy HREPLOBJ objects. The function CopyObject does this by copying a REPLOBJECT from one location to another (Listing 17.21).

Listing 17.21 *IReplStore:: CopyObject implementation*

```
STDMETHODIMP_(BOOL) CActiveSyncEg::CopyObject(
        HREPLOBJ hObjSrc, HREPLOBJ hObjDest)
{
  LPREPLOBJECT lpRepObjSrc = (LPREPLOBJECT)hObjSrc;
  LPREPLOBJECT lpRepObjDest = (LPREPLOBJECT)hObjDest;
  *lpRepObjDest = *lpRepObjSrc;
  return TRUE;
}
```

Finally, IsValidObject is called when ActiveSync needs to determine whether the HREPOBJ still refers to a valid item or folder. For folders, Listing 17.22 simply checks that the REPLOBJECT uType is RT_FOLDER, returning NOERROR if it is, or RERR_CORRUPT to indicate the object is no longer valid.

Listing 17.22 *IReplStore:: IsValidObject implementation*

```
STDMETHODIMP CActiveSyncEg::IsValidObject(
    HREPLFLD hFld, HREPLITEM hItem, UINT uFlags )
{
  LPREPLOBJECT lpRepObj;

  if(hFld != NULL)
  {
    lpRepObj = (LPREPLOBJECT)hFld;
    if(lpRepObj->uType == RT_FOLDER)
      return NOERROR;
    else
      return RERR_CORRUPT;
  }

  if(hItem!= NULL)
  {
    lpRepObj = (LPREPLOBJECT)hItem;
```

```
    if(lpRepObj->uType != RT_ITEM)
      return RERR_CORRUPT;
    NOTE aNote;
    // attempt to find the item
    if(m_ListDB.FindNote(&lpRepObj->ftCreated,
          &aNote))
      return NOERROR;
    else
      return RERR_OBJECT_DELETED;
  }
  return NOERROR;
}
```

For items, a check needs to be made that `uType` is `RT_ITEM`. Additionally, the function needs to check that the item referred to by the `HREPLFLD` is still present in the store. Calling `FindNote` (implemented in `db.cpp`) does this.

HREPLITEM Synchronization

`HREPLITEM` objects are maintained to allow ActiveSync to track changes to objects either on the desktop or the Windows CE device. The provider must implement the functions listed in Table 17.6.

Table 17.6 *Functions to Synchronize HREPLITEM items*

Function	Description
`CompareItem`	Do two `HREPLITEM`s refer to the same item?
`IsItemChanged`	Has the `HREPLITEM` changed?
`IsItemReplicated`	Is the `HREPLITEM` to be replicated?
`UpdateItem`	Update the information in the `HREPLITEM` based on what is stored in the desktop database.

In general, at least two pieces of information are held in a `HREPLITEM` to enable these functions:

1. The unique identifier, so items can be compared. In `REPLOBJECT` this is `ftCreated`, the `FILETIME` timestamp of when the item was created.
2. The last modification identifier, so changes to items can be tracked. In `REPLOBJECT` this is the `ftModified FILETIME` timestamp.

`CompareItem` is called when ActiveSync needs to know whether two `HREPLITEM`s refer to the same item in the store. This would be called, for example, when an item from the Windows CE device is updated and ActiveSync needs to find the corresponding item in the store. `CompareItem` (Listing 17.23) simply passes the `ftCreated` members to `CompareFileTime`, which returns 0 if the `FILETIME`s are the same, −1 if the first is earlier, or 1 if the first is later.

Listing 17.23 *IReplStore:: CompareItem implementation*

```
STDMETHODIMP_(int) CActiveSyncEg::CompareItem(
    HREPLITEM hItem1, HREPLITEM hItem2 )
{
  LPREPLOBJECT lpRepObj1 = (LPREPLOBJECT)hItem1;
  LPREPLOBJECT lpRepObj2 = (LPREPLOBJECT)hItem2;
  int nRet = CompareFileTime(&lpRepObj1->ftCreated,
      &lpRepObj2->ftCreated);
  return nRet;
}
```

HREPLITEM maintains a modification timestamp that may be different from the item in the database. For example, the database item may have changed since the time the HREPLITEM was created. ActiveSync calls IsItemChanged to determine if this is the case (Listing 17.24).

IsItemChanged passes in two HREPLITEM objects. If both are non-NULL, the function compares the two ftModified members in the REPLOBJECT structures pointed to by hItem and hItemComp. If hItemComp is NULL, IsItemChanged compares the ftModified for hItem with the modification time of the item in the database. This is obtained by finding the item in the database using FindNote (db.cpp).

Listing 17.24 *IReplStore:: IsItemChanged implementation*

```
STDMETHODIMP_(BOOL) CActiveSyncEg::IsItemChanged(
    HREPLFLD hFld,
    HREPLITEM hItem,
    HREPLITEM hItemComp )
{
  LPREPLOBJECT lpRepObj1 = (LPREPLOBJECT)hItem;
  if(hItemComp != NULL)
  {
    LPREPLOBJECT lpRepObj2 = (LPREPLOBJECT)hItemComp;
    return CompareFileTime(
      &lpRepObj1->ftModified,
      &lpRepObj2->ftModified);
  }
  else
  {
    // need to compare this object with the
    // one in the .DAT file
    NOTE aNote;
    if(m_ListDB.FindNote(
        &lpRepObj1->ftCreated, &aNote))
      return CompareFileTime(
```

```
            &lpRepObj1->ftModified,
            &aNote.ftLastUpdate);
        else
        {
          MessageBox(NULL,
            _T("Could not find record for \
              IsItemChanged"), NULL, 0);
          return FALSE;
        }
      }
    }
```

Generally, all items in the store are synchronized. However, there are times when you may want to filter the items being synchronized—for example, you may only want to filter appointments from the last two weeks. The function `IsItemReplicated` is passed a `HREPLITEM`, and returns `TRUE` if the item is to be synchronized (Listing 17.25).

Listing 17.25 *IReplStore:: IsItemReplicated implementation*

```
STDMETHODIMP_(BOOL) CActiveSyncEg::IsItemReplicated(
    HREPLFLD hFld, HREPLITEM hItem )
{
  return TRUE;
}
```

Implementing the Desktop IReplObjHandler COM Interface

The desktop application must implement the same `IReplObjHandler` interface functions as the device, but the implementations will typically be different. In the example the class `CDataHandler` in the file `ReplObjHandler.cpp` implements the `IReplObjHandler` interface.

IReplObjHandler:: Setup

ActiveSync calls this function before any item is received or sent. This provides an opportunity to perform initialization. A pointer to a `REPLSETUP` structure is passed in; this provides information such as the direction of the transfer.

The `Setup` function will normally save a pointer to the `REPLSETUP` structure for future use. Since ActiveSync is multithreaded, it is possible that a read (outgoing transfer) occurs at the same time as a write (incoming transfer). Therefore, the `IReplObjHandler` class has two members, `m_pReadSetup` and `m_pWriteSetup` to store separate pointers (Listing 17.26). The pointer will be used later in `GetPacket` and `SetPacket`.

Microsoft Windows CE

Nick Grattan • Marshall Brain

Windows® CE 3.0

APPLICATION PROGRAMMING

- Beyond the user interface to hard-core programming
- Full-scale networking and enterprise computing
- Global communications from Pocket PC®s
- All the new features of Windows CE 3.0

MICROSOFT® TECHNOLOGIES SERIES

- Networking
- ✔ Programming
- BackOffice
- Administration
- Certification

LEVEL

Intermediate ——→ Advanced

Advanced techniques for serious Windows CE programmers

Get beyond user interface programming and discover the behind-the-scenes operating system facilities that will let you make the most of the new features in Windows CE 3.0. This hot technology lets you control Pocket PCs, hand-held PCs, and the embedded devices in hundreds of commercial products. Learn the lean and mean techniques that keep your programs humming on portable devices with limited memory, and the key data storage methods that make them possible. Master the communications protocols that keep Windows CE devices in contact with desktop computers and the Internet. In addition:

- Build and run applications in Visual C++® 6.0 and eMbedded Visual C++ 3.0
- Use the Windows CE API and Microsoft® Foundation Classes
- Communicate via HTTP, TCP/IP, sockets, remote access, and telephony
- Access standard Windows CE databases and Microsoft SQL Server for Windows CE
- Interface between desktop systems and Windows CE devices

This book is for serious developers with real programming experience. Besides familiarity with Windows CE devices and general Windows API programming, a basic knowledge of C and C++ is needed to understand the code samples.

Included

About the CD-ROM

The accompanying CD-ROM contains all the code examples from the book, as well as a fully searchable index of all the book's examples, programs, and tutorials. The CD-ROM also contains a complete working copy of eMbedded Visual C++ 3.0 and Pocket PC® SDK.

About the Authors

NICK GRATTAN is co-founder and Technical Director at Software Paths Limited (www.SoftwarePaths.com), a Dublin, Ireland, based mobile solutions specialist.

MARSHALL BRAIN is the founder and CEO of How Stuff Works, an educational content company that explains complex subjects in simple terms. He is the author of 12 books, including *Win32 API Programming*. He holds degrees in Engineering and Computer Science and has been recognized for his teaching excellence by the prestigious Academy of Outstanding Teachers.

PRENTICE HALL
Upper Saddle River, NJ 07458
www.phptr.com

$49.99 U.S.
$75.00 Canada

0 76092 00897 2

ISBN 0-13-025592-0

9 780130 255921 90000