

# Chapter 14

## The Remote API (RAPI)

The last chapter discussed Web Services, one option for connecting CF-based devices to desktop (and server) systems. This chapter covers the Remote API, a second option for making the device-to-desktop connection for devices that are connected using ActiveSync. [\[Comment 21cs.1\]](#)

What Is ActiveSync? .....	2
Partners And Guests .....	3
ActiveSync Limitations .....	4
One Active Connection .....	4
No Emulator Support .....	4
No Synchronization Between Devices .....	4
No Synchronization with Servers .....	5
ActiveSync Compared to Web Services .....	5
ActiveSync Programming Interfaces .....	5
Should You Build Managed-Code ActiveSync Applications? .....	6
The Remote API (RAPI) .....	8
Available RAPI Functions .....	9
Building .NET ActiveSync Applications .....	12
Remote API Startup and Shutdown .....	12
CeRapiInit versus CeRapiInitEx .....	13
Shutdown with CeRapiUninit .....	13
Two Approaches To Startup .....	13
Defensive Coding .....	21
Accessing the Object Store .....	21
Using RAPI to Access Device Files .....	22
FindPrograms – Using RAPI to Find Files in the Device File System .....	24
FindProgramsFaster – Speeding up a File Search .....	29
Remote Access to Device Registry Entries .....	34
Remote Access to Device Property Databases .....	43
Detecting Changes in Device Connection State .....	46
The Auto-Start Method .....	46
The Callback Method .....	47
Loading Programs and DLLs .....	52
Running Device-Side Programs .....	52
Loading Device-Side DLLs .....	53
Building a DLL Using Embedded Visual C++ .....	55
Calling a DLL function with CeRapiInvoke .....	57
Conclusion .....	58

This chapter discusses the Remote API, an extension to the Win32 API on desktop versions of Microsoft Windows. The RAPI functions provide a mechanism for code running on the desktop to access Windows CE-powered devices which are connected to the desktop via ActiveSync. Most of the RAPI functions involve accessing the object store of a connected device, although there are a host of other functions that provide some interesting ways to access a Windows CE system from a desktop system.

Because RAPI is dependent on ActiveSync, it makes sense that we start this chapter with an exploration of ActiveSync.

---

## What Is ActiveSync?

ActiveSync is a desktop-oriented set of services for connecting a Windows CE device to a desktop system running any recent version of Microsoft Windows (Microsoft Windows 98 or later). The purpose of an ActiveSync connection is to (1) move data from a device to a desktop system, (2) move data from a desktop system to a device, (3) install an application, (4) de-install an application, or (5) allow the device to access a network using the desktop computer's network card and ActiveSync's *Pass Through* support. [\[Comment 21cs.2\]](#)

### ActiveSync *Pass Through*

When ActiveSync 3.5 or later is on the desktop, a docked Pocket PC 2002 has access to ActiveSync *Pass Through*. Andreas Sjöström, of Business Anyplace ([www.businessanyplace.net](http://www.businessanyplace.net)) in Sweden, wrote an article on this subject on Microsoft's Pocket PC site, which you can read: <http://www.microsoft.com/mobile/developer/technicalarticles/passthrough.asp>. From a development point of view, Andreas makes two key points about what protocols *Pass Through* supports and which ones it does not. [\[Comment 21cs.3\]](#)

In brief, *Pass Through* supports HTTP, HTTPS, IMAP and POP3. It does not support UDP, ICMP, or PPTP. [\[Comment 21cs.4\]](#)

Kenny Goers suggests that FTP works well in the *Pass Through* layer as well. Kenny is a developer at Magenic ([www.magenic.com](http://www.magenic.com)), and the moderator of the Yahoo Group where Paul Yao likes to hang out and offer friendly advice and the occasional bad pun. For details on the list, browse to <http://groups.yahoo.com/group/windowsce-dev>. [\[Comment 21cs.5\]](#)

ActiveSync itself is installed on desktop systems, not on Windows CE devices. Devices like the Pocket PC come packaged with a docking cradle that serves both as a battery charger and also to provide the physical connection between a device and a serial or USB connection. When a device is put into its docking cradle, ActiveSync wakes up and establishes a connection. [\[Comment 21cs.6\]](#)

When you purchase a Pocket PC or other Windows CE-based device, you normally also find a CD containing ActiveSync for installation on your desktop system. ActiveSync is also available for download from the Microsoft web site (<http://www.microsoft.com/mobile/pocketpc/downloads/activesync35.asp>), and it can be used to create connections to custom Windows CE based devices as well. [\[Comment 21cs.7\]](#)

**Table 14-1 ActiveSync sample programs in this chapter** [\[Comment 21cs.8\]](#)

Name	Description
RapiStartup	Simple RAPI program with startup and shutdown and no threads.
FindPrograms	Searches device file system for *.exe files using CeFindFirstFile and CeFindNextFile approach.
FindProgramsFaster	Faster program search sample using CeFindAllFiles.

<i>Name</i>	<i>Description</i>
FindProgramsOnConnect	FindProgramsFaster with IDCCMan connect detection to search for program files automatically when a connection is detected.
RegShowStartup	Queries device registry to show programs and DLLs that are automatically started at CE system cold boot.
ShowDatabases	Queries device for mounted property database volumes and property databases.
YaoDurant.Win32.Rapi	Library containing P/Invoke wrappers for RAPI.DLL
SimpleBlockModeInvoke	Win32 DLL that demonstrates block mode call to CeRapiInvoke.
CallDeviceDll	Test program that calls SimpleBlockModeInvoke.dll

When a password has been established on a Windows CE device, ActiveSync asks for a password before allowing access to a device. Windows CE devices support two kinds of passwords: a simple four digit number, as well as a longer alphanumeric value. Users set the password in the Windows CE Control Panel. [\[Comment 21cs.9\]](#)

ActiveSync treats a Pocket PC – or other Windows CE-powered devices – as a remote peripheral. This represents a very different approach to Windows CE devices from the other chapters in this book. For this reason, most of the code in this chapter runs on the desktop and not on a device. Two sample programs run on a device, but these are downloaded and run from a desktop program. This chapter's sample programs are summarized in table 14-1. The C# versions are located on the book CD under the `\YaoDurant\CS\ActiveSync` directory. You can find the VB versions under the `\YaoDurant\VB\ActiveSync` directory. [\[Comment 21cs.10\]](#)

### Partners And Guests

When a device connects to a desktop, ActiveSync determines whether the desktop and the device have an established partnership. If so, ActiveSync performs a set of synchronization tasks. If there is no partnership, the user can establish a partnership or connect as a guest. Full synchronization services are only available to partner devices and not to guests. [\[Comment 21cs.11\]](#)

Either partner or guest devices can do the following: [\[Comment 21cs.12\]](#)

- 1 Manual transfer of files between the desktop and the device using the desktop Windows Explorer. [\[Comment 21cs.13\]](#)
- 2 Add or remove – that is, to install or de-install – programs [\[Comment 21cs.14\]](#)
- 3 Back up and restore device data [\[Comment 21cs.15\]](#)
- 4 Connect to an intranet or the Internet using ActiveSync Pass Through [\[Comment 21cs.16\]](#)

Devices with partnerships can do the following additional tasks:

- 1 Synchronize files. Files in specially marked folder get copied from the device to the desktop, or from the desktop to the device depending on which was more recently modified. [\[Comment 21cs.17\]](#)
- 2 Synchronize databases. When a partner device connects to a desktop system, ActiveSync runs all active database synchronization. There are several built-in database synchronization objects, including one for the Pocket Outlook calendar, contact list, inbox, and task list. Applications

can also install custom database synchronization objects. [\[Comment 21cs.18\]](#)

A device can establish a local connection to a desktop system using any of several mechanisms: via USB port, serial port, Infrared (IrDA) port, or via Bluetooth. Once a partnership has been established between a device and a desktop, remote connections can be established using your choice of dial-up modem, a hard-wired network connection, or a wireless Wi-Fi network connection. [\[Comment 21cs.19\]](#)

### ActiveSync Limitations

Before digging into ActiveSync, we ought to point out its limitations: [\[Comment 21cs.20\]](#)

- 1 ActiveSync only supports one active connection at a time. [\[Comment 21cs.21\]](#)
- 2 ActiveSync cannot access the Windows CE emulator. [\[Comment 21cs.22\]](#)
- 3 ActiveSync does not synchronize between devices. [\[Comment 21cs.23\]](#)
- 4 ActiveSync does not synchronize with servers. [\[Comment 21cs.24\]](#)

#### *One Active Connection*

A given Windows CE device can have a partnership with at most two different desktop systems. Desktop systems, on the other hand, can have partnerships with an unlimited number of devices. ActiveSync itself, however, can only handle one device connection at a time. Windows is multi-tasking, but ActiveSync is not. [\[Comment 21cs.25\]](#)

This makes ActiveSync suitable for some scenarios, and inappropriate for others. ActiveSync works well when there is a 1:1 mapping between devices and desktops. ActiveSync does not work as well with a many-to-one relationship between devices and desktop systems. For example, if you have a large number of devices that leave in the morning and come back in the evening, ActiveSync is probably not your best choice. It is for this reason that Microsoft SQL Server CE, a database that runs on Windows CE, does not use ActiveSync for its device-to-database connectivity. Because it does not rely on ActiveSync, SQL Server CE allows multiple Windows CE device to simultaneously synchronize with a single central database server. [\[Comment 21cs.26\]](#)

#### *No Emulator Support*

The lack of emulator support means that to build and run ActiveSync applications, you must have an actual Windows CE device. If you plan to spend any time working on ActiveSync code, we suggest you make the investment in a network card for your Windows CE device. The reason is simple: network hardware is relatively inexpensive, and it allows for much faster development and debugging. In fact, this suggestion is not just limited to ActiveSync development – but for all your device-side development and testing. Network download and debugging – whether by wired network or wireless – is just plain faster than serial or USB port connections. [\[Comment 21cs.27\]](#)

#### *No Synchronization Between Devices*

ActiveSync is not a general purpose synchronization mechanism, but is specifically designed to connected devices with desktop – and not with any desktop, but desktops running Microsoft Windows 98 and later. Supported versions of Microsoft Windows include Windows 98, Windows Me, Windows NT (Service Pack 6), Windows 2000, Windows XP, and the .NET Server. [\[Comment 21cs.28\]](#)

### **NOTE ActiveSync Backup and Restore**

While ActiveSync does not provide support for synchronizing between devices, it does allow you to backup the object store from one device, and restore to another Windows CE device. The end result is two identical devices. For users of Windows CE-based devices, this ability to backup and restore can help safeguard against data loss when a device is lost or damaged. [\[Comment 21cs.29\]](#)

#### *No Synchronization with Servers*

ActiveSync does not synchronize Windows CE devices with servers. There are, however, server-side systems that communicate with Windows CE-powered devices. These are summarized in the accompanying sidebar, *Connecting Devices to Servers*. Those server side systems do not use ActiveSync, but rather support direct connections to devices through TCP/IP. [\[Comment 21cs.30\]](#)

#### **ActiveSync Compared to Web Services**

Since ActiveSync and Web Services both support desktop-to-device connectivity, you might at this point wonder how they compare. Both provide Remote Procedure Call (RPC) support; however ActiveSync is a more specialized form of RPC while Web Services provide a more general purpose type of RPC. [\[Comment 21cs.31\]](#)

ActiveSync differs from Web Services in that the locus of control for a web service is at the client – it is, after all, the device that calls the web service. With ActiveSync, on the other hand, the locus of control is on the desktop as that is where your code primarily runs. [\[Comment 21cs.32\]](#)

ActiveSync is a desktop-oriented programming interface, which provides a very simple remote procedure call mechanism for accessing the object store of a Windows CE device. Use it when your usage scenario involves having a home base for a roaming device. The home base – the desktop system – provides the place to install software and synchronize data. [\[Comment 21cs.33\]](#)

Web services, on the other hand, are a general purpose remote procedure call (RPC) mechanism that allows a device to access a standard (meaning here the SOAP / XML / HTTP-based standards) web service. You will likely use web services to call out to third-party services to fetch phone numbers, the weather, travel information, etc. [\[Comment 21cs.34\]](#)

For the most part, then, the uses of ActiveSync and Web Services are separate and distinct – use ActiveSync to "phone home", and Web Services to "call out." The only real overlap is when you implement Web Services on the PC where a Windows CE device has a partnership. In that case, the decision factor comes down to whether you want to implement something that is device-driven or desktop-driven. For device-driven operations, implement a Web Service; for desktop-driven operations, use ActiveSync. [\[Comment 21cs.35\]](#)

#### **ActiveSync Programming Interfaces**

ActiveSync provides a set of Win32 functions and COM interfaces to support its various services. This chapter focuses on the Remote API (RAPI) and on the Device Connection Manager. Table 14-2 shows these two, along with other ActiveSync function sets that are available for your use. [\[Comment 21cs.36\]](#)

**Table 14-2 ActiveSync Application Programming Interfaces** [\[Comment 21cs.37\]](#)

<i>ActiveSync API</i>	<i>Description</i>	<i>Build Files</i>	<i>Runtime Files</i>
Remote API	A set of 78 C-callable functions for accessing the	RAPI.H	RAPI.DLL

	file system, registry, property databases, and for supporting various application setup activities	RAPI.LIB	
File Filter	COM-based interface for converting whole files as they are copied from the device to the desktop, or from the desktop to a device.	REPLFILT.H	
Device Connection Manager Notification	COM-based interface for receiving notifications about when ActiveSync connections start and stop	DCCOLE.H	RAPI.DLL
Registry Service Functions	Set of helper functions to read & write desktop registry settings in support of ActiveSync.	CEUTIL.H CEUTIL.LIB	CEUTIL.DLL
Database Synchronization	COM-based interfaces called by ActiveSync to keep device-side and desktop-side databases synchronized.	CESYNC.H	

### Should You Build Managed-Code ActiveSync Applications?

All of the ActiveSync libraries listed in table 14-2 are unmanaged code libraries, built with either C-callable exported functions or COM components. That being the case, you might wonder whether you can write ActiveSync applications in managed code, and whether you should. [\[Comment 21cs.38\]](#)

Can you use managed code? Yes. P/Invoke support lets you call C-callable functions, and COM Interop lets you call into and implement managed-code interfaces. To access ActiveSync services from managed code, you need a set of P/Invoke wrappers – which is what we have done for you. You can find those wrappers on the CD which accompanies this book. (Or, you can use the P/Invoke Wizard – discussed in detail in Chapter 4 – to build your own P/Invoke wrappers.) [\[Comment 21cs.39\]](#)

While the Compact Framework does not support COM Interop, the Desktop Framework does have that support. So any desktop-side ActiveSync component that requires COM can be easily created as managed code; but any device-side ActiveSync component that requires COM cannot be written as managed code, but instead must be written in native (a.k.a. “unmanaged” or “Win32” code). [\[Comment 21cs.40\]](#)

The next question is whether you should take this approach. There are myriad benefits that you as a programmer get from managed code, which we discuss in Part I of this book. Because of all its benefits, we are partial to managed code. If you are also, then the good news is that most ActiveSync features can be accessed in managed code. [\[Comment 21cs.41\]](#)

A reason not to use managed code is because they rely on the .NET runtime libraries. While the Desktop Framework is compatible with Windows 98 and later, it was not released until January of 2002 and so is not yet widely available. Microsoft provides a mechanism for distributing the desktop framework – a file named `dotnetfx.exe`. This 20MB file installs the .NET Framework, and you can include this on a CD that you supply your customers. Details are available on the Microsoft web site:

<http://msdn.microsoft.com/library/en-us/dnnetdep/html/redistdeploy.asp>. [\[Comment 21cs.42\]](#)

### **(SideBar) Connecting Devices to Servers**

Microsoft's commitment to mobile & embedded devices can be seen in the energy devoted to create and enhance Windows CE, which first shipped in 1996. Version 4.1 started shipping in the summer of 2002. In

recent years, Microsoft has started adding Windows CE device support to its various server-side products. [\[Comment 21cs.43\]](#)

**SQL Server 2000 Windows CE Edition.** Known also as *SQL Server CE*, this is a single-user, stand-alone database engine for Windows CE devices. SQL Server CE supports two mechanisms for connecting to and synchronizing device data with server data – Merge Replication and Remote Data Access (RDA). The connection between a Windows CE device and a SQL Server database is not through ActiveSync, but instead is an HTTP-based connection. [\[Comment 21cs.44\]](#)

The typical scenario for a SQL Server CE user is a salesperson with a Pocket PC and a mobile phone, or the two combined together, as in the Pocket PC 2002 Phone Edition. Such a user can remotely synchronize the mobile database with the central office database. In between sales calls, the mobile SQL Server CE database can connect, via the mobile phone network, to the central server to both upload orders and download status information. For details, see the June 2001 edition of MSDN Magazine, *SQL Server CE: New Version Lets You Store and Update Data on Handheld Devices*, by Paul Yao and David Durant. This article is available online at:

<http://msdn.microsoft.com/msdnmag/issues/01/06/sqlce/default.aspx>.

[\[Comment 21cs.45\]](#)

**Microsoft Mobile Internet Toolkit (MMIT).** The Mobile Internet Toolkit was built to support web-based applications for mobile and handheld devices. It allows developers to add mobile device support to ASP .NET-based web sites. A separate toolkit is needed because devices – PDAs and cell phones – have smaller displays than desktop systems. Devices often have browsers that support simpler markup language than what is found in desktop browsers. MMIT was built to support all of these differences. As of this writing, MMIT supports three markup languages: HTML 3.2, Compact HTML (cHTML), and Wireless Markup Language (WML). For details, see our article from the November 2002 edition of MSDN Magazine, *Microsoft Mobile Internet Toolkit Lets Your Web Application Target Any Device Anywhere*, by Paul Yao and David Durant. This article is available online at:

<http://msdn.microsoft.com/msdnmag/issues/02/11/MMIT/default.aspx>.

[\[Comment 21cs.46\]](#)

**Mobile Information Server.** Microsoft created the Mobile Information Server (MIS) to allow device-to-server connectivity for Exchange Server, Microsoft's email server system. MIS allows remote access from a Pocket PC, or other supported Windows CE device, to data that a user might access from the desktop using Microsoft Outlook. A device user can log in and remotely access email, task lists, contact lists, and calendars. Details are available at the Microsoft web site:

<http://www.microsoft.com/miserver>. [\[Comment 21cs.47\]](#)

**Systems Management Server.** Another Microsoft server-side system called *Systems Management Server (SMS)* allows for centralized control for downloading and installing software. As of this writing, there have been no public announcements that SMS will support Windows CE devices.

But that would certainly be a good idea, and perhaps in a future version of SMS, this support will become available. [\[Comment 21cs.48\]](#)

---

## *The Remote API (RAPI)*

The Windows CE Remote API is a specialized remote procedure call (RPC) facility. We call it a "remote procedure call" API because RAPI functions cause remote function calls on connected devices. It is "specialized" because you can only call a limited subset of device-side functions. Most RAPI functions provide access to a device's object store and device-side file systems. As we describe in Chapter 16, the *object store* is the permanently mounted RAM-based storage area that contains the built-in file system, the system registry, and property databases. This is not the only storage available, however, and RAPI also lets you access whatever installable file system is present to support removable Compact Flash cards, Smart Media cards, disk drives, etc. [\[Comment 21cs.49\]](#)

### **Remote API and .NET Remoting** [\[Comment 21cs.50\]](#)

If you have worked with the desktop .NET Framework, you might have heard about *.NET Remoting* and be wondering about its relationship to the Remote API. Aside from similar names, these two technologies have nothing in common. [\[Comment 21cs.51\]](#)

The ability to access the object store means that a RAPI program can access any stored data. You can, for example, open a file and copy part of it – or all of it – from the device to the desktop. You could open the system registry and create new keys, or read and write values on existing keys. You have complete access to the property databases in the object store, so that you can create a database, delete a database, add or remove database records, and read or write individual property values. [\[Comment 21cs.52\]](#)

The Remote API is a set of functions that are exactly like the Win32 functions used to access files, registry entries, and CE databases. The only difference is that each of the functions has a slightly different name – a prefix of "Ce." For example, the Win32 function to open a file is `CreateFile`; its RAPI equivalent is `CeCreateFile`. Once a file is opened, you read a file's contents by calling `CeReadFile`, and close the file by calling `CeCloseHandle`. This is different from the approach we took to file access in Chapter 15, where we discuss using `System.IO` classes. And instead of ADO .NET classes, access to property databases is through a set of C-callable functions with names like `CeCreateDatabase` and `CeWriteRecordProps`. [\[Comment 21cs.53\]](#)

### **NOTE RAPI Names for Property Database Functions**

CE property databases were created specifically for Windows CE, and already have a `Ce` prefix in their names. So device-side database functions have the same name as their RAPI function counterpart. In other words, `CeCreateDatabase` is both a device-side function and a RAPI function. [\[Comment 21cs.54\]](#)

Most RAPI functions access the object store, but there are also non-object store RAPI functions. Two interesting functions are `CeCreateProcess` and `CeRapiInvoke`. `CeCreateProcess` lets you start running a device-side program. `CeRapiInvoke` remotely loads and calls a function in a device-side dynamic link library (DLL). These functions

are very useful, giving you wide latitude to do almost anything on a device. You could, for example, download an executable file to a device, and then start it running. With these two functions, there is almost nothing that you cannot do. [\[Comment 21cs.55\]](#)

You could, for example, copy a Compact Framework program onto a Windows CE device and start it running. You could download a Win32 dynamic link library and call a function. You can then keep those files on the target device, or remove them – perhaps to save device-side storage space. [\[Comment 21cs.56\]](#)

### Available RAPI Functions

We divide the 78 RAPI functions into eight groups. Table 14-3 provides details of each group. The groups are: [\[Comment 21cs.57\]](#)

- RAPI Support
- Run Program / Load DLL
- File System
- Registry
- CE Property Database
- Object Store Queries
- System Information
- Remote Windowing

Most RAPI functions have an equivalent Win32 function in Windows CE. In most cases, the name of the RAPI function is created by adding a prefix of "Ce" to the Win32 function name. A few RAPI functions have no Win32 equivalent, such as those used for RAPI management and support. In two cases – `CeFindAllFiles` and `CeFindAllDatabases` – functions were created specifically to boost performance. These two functions access a user-defined subset in a query, allowing for faster queries of available files or databases. [\[Comment 21cs.58\]](#)

**Table 14-3 Available RAPI Functions** [\[Comment 21cs.59\]](#)

<i>Group 1: RAPI Support</i>	
<i>RAPI Function</i>	<i>Description</i>
<code>CeRapiInit</code>	Blocking initialization function.
<code>CeRapiInitEx</code>	Non-blocking initialization function – returns a Win32 event handle that gets signaled when a connection is complete.
<code>CeRapiUninit</code>	Cleanup function for <code>CeRapiInit</code> and <code>CeRapiInitEx</code> .
<code>CeRapiGetError</code>	Query for RAPI-specific errors.
<code>CeGetLastError</code>	Query for non-RAPI device-side errors (equivalent to <code>GetLastError</code> ).
<code>CeRapiFreeBuffer</code>	Free buffer allocated by various functions, including <code>CeFindAllFiles</code> and <code>CeFindAllDatabases</code> .
<i>Group 2: Run Program / Load DLL</i>	
<i>RAPI Function</i>	<i>Description</i>
<code>CeCreateProcess</code>	Start a program running on the Windows CE device.
<code>CeRapiInvoke</code>	Call a named function in a named Win32 DLL on a Windows CE device. Does not support generic remote procedure calls to DLLs, since function called must take a specific set of parameters.

<i>Group 3: File System</i>	
<i>RAPI Function</i>	<i>Description</i>
CeCloseHandle	Close a file opened with a call to CeCreateFile (or a database opened with CeOpenDatabaseEx).
CeCopyFile	Copy a file on the device to another location on the device (does not copy between device and desktop system).
CeCreateDirectory	Create a directory in the file system of a device.
CeCreateFile	Open a file on the device, optionally creating it at the same time.
CeDeleteFile	Delete a file on the device file system.
CeFindAllFiles	Special RAPI function to search for all files in a given directory that match a specific criteria, returning a subset of the data returned by the normal file enumeration functions. This reduction in data causes this function to run faster than the generic file enumeration functions.
CeFindClose	Cleanup function for file enumeration begun with call to CeFindFirstFile.
CeFindFirstFile	Starts file enumeration in a given directory for specific search criteria.
CeFindNextFile	Continues file enumeration begun with call to CeFindFirstFile.
CeGetFileAttributes	Query attribute of a file in the device file system.
CeGetFileSize	Query file size.
CeGetFileTime	Query creation, last accessed and last modified date and time.
CeGetSpecialFolderPath	Get path name for shell file system folders.
CeGetTempPath	Get path to temporary directory.
CeMoveFile	Rename an existing file or directory on a device.
CeReadFile	Read a number of bytes from a file opened with a call to CeCreateFile.
CeRemoveDirectory	Delete an existing empty directory.
CeSetEndOfFile	Truncate file at the current position of the file pointer.
CeSetFileAttributes	Modify the file attribute flags to enable or disable the following bits: archive, hidden, normal, readonly, system, and temporary.
CeSetFilePointer	Move file pointer in a file opened with CeCreateFile.
CeSetFileTime	Modify the directory entry for a file on a device to change the associated created, last accessed, or modified time.
CeSHCreateShortcut	Create a program shortcut on the remote device.
CeSHGetShortcutTarget	Query the path stored in an existing shortcut.
CeWriteFile	Write a number of bytes to a file opened with a call to CeCreateFile.
<i>Group 4: Registry</i>	
<i>RAPI Function</i>	<i>Description</i>
CeRegCloseKey	Close a registry key opened with a call to either CeRegOpenKeyEx or CeRegCreateKeyEx.
CeRegCreateKeyEx	Create a new registry key – or open an existing registry key.
CeRegDeleteKey	Delete a named sub-key and the associated value in the registry. If the named sub-key has sub-keys under it, the call fails.
CeRegDeleteValue	Delete one value from the specified registry node.
CeRegEnumKeyEx	Enumerate the sub-keys of an indicated registry node.
CeRegEnumValue	Enumerate the values in an indicated registry node.
CeRegOpenKeyEx	Open a registry node.

CeRegQueryInfoKey	Query details about a give registry node, including number of sub-keys and the length of the longest sub-key name.
CeRegQueryValueEx	Fetch a value from a registry node.
CeRegSetValueEx	Write a value to a registry node.
<i>Group 5: CE Property Database</i>	
<i>RAPI Function</i>	<i>Description</i>
CeCloseHandle	Close a database opened with CeOpenDatabaseEx (or a file opened with CeCreateFile).
CeCreateDatabase	Create a new property database. Obsolete – use CeCreateDatabaseEx instead.
CeCreateDatabaseEx	Create a new property database with an associated GUID.
CeDeleteDatabase	Delete a database by object ID (OID) value.
CeDeleteDatabaseEx	Delete a database by object ID and GUID.
CeDeleteRecord	Delete a database record.
CeEnumDBVolumes	Enumerate all mounted property database volumes.
CeFindAllDatabases	Special RAPI database query function that returns a subset of the data returned by the general purpose database enumeration functions. The reduction in requested data causes this to run faster than the general purpose enumeration functions.
CeFindFirstDatabase	Start enumerating all property databases in the object store.
CeFindFirstDatabaseEx	Start enumerating all property databases in the system, or on the indicated database volume.
CeFindNextDatabase	Continue enumeration started with CeFindFirstDatabase.
CeFindNextDatabaseEx	Continue enumeration started with CeFindFirstDatabaseEx
CeFlushDBVol	Force a save of database changes from program memory to storage.
CeMountDBVol	Issue a mount request for a database volume.
CeOpenDatabase	Open a database. Obsolete – use CeOpenDatabaseEx instead.
CeOpenDatabaseEx	Open a database by name or by GUID.
CeReadRecordProps	Read a database record.
CeReadRecordPropsEx	Read a database record, optionally allocating memory from a provided Win32 heap.
CeSeekDatabase	Set the current database record based on selection criteria.
CeSetDatabaseInfo	Modify various database attributes, including name, type, and sort order.
CeSetDatabaseInfoEx	Modify various database attributes, including name, type, and sort order, as well as the GUID.
CeUnmountDBVol	Un-mount a database volume mounted with an earlier call to CeMountDBVol
CeWriteRecordProps	Write a record to the database.
<i>Group 6: Object Store Queries</i>	
<i>RAPI Function</i>	<i>Description</i>
CeOidGetInfo	Query the type of record located within the object store for a given object id (OID) value.
CeOidGetInfoEx	Query the type of record in the object store, or for any mounted database volume, for a given object id (OID) value.
CeGetStoreInformation	Query total size and free space for the object store.
<i>Group 7: System Information</i>	
<i>RAPI Function</i>	<i>Description</i>

CeCheckPassword	Compare an entered password to the system password, as set in the Windows CE control panel. Password checking is not required for an application to RAPI, because – when password protection is enabled – ActiveSync requires the user to enter a valid password before a RAPI connection can be established. You would likely only check the password as a double-check for sensitive operations.
CeGetDesktopDeviceCaps	Query the graphics capabilities of a remote device (this is equivalent to the GetDeviceCaps graphic capability query function).
CeGetSystemInfo	Query the CPU and memory architecture of a remote device.
CeGetSystemMetrics	Query the size of elements on the user-interface of the remote system, including the width and height of window borders, icons, scroll bars, etc.
CeGetSystemPowerStatusEx	Query whether wall power is available and battery status.
CeGetVersionEx	Query Windows CE operating system version information.
CeGlobalMemoryStatus	Query available memory load, amount of physical memory, and available virtual memory.
<i>Group 8: Remote Windowing</i>	
<i>RAPI Function</i>	<i>Description</i>
CeGetClassName	Query the window class name for a given window handle.
CeGetWindow	Walk the window hierarchy of the remote device.
CeGetWindowLong	Query state information for an indicated window.
CeGetWindowText	Query the text of an indicated window.

### Building .NET ActiveSync Applications

Because ActiveSync applications run on the desktop, when you build them in Visual Studio .NET, you do not create a "Smart Device Application." You pick instead any of the various desktop project types. You can build ActiveSync applications either as managed (.NET) applications, or as unmanaged (Win32) applications. This book does not cover the Win32 approach, since it is covered elsewhere, such as in Doug Boling's book, *Programming Microsoft Windows CE* (Microsoft Press, 2001). This chapter focuses instead on building managed code ActiveSync applications. [\[Comment 21cs.60\]](#)

If you plan to call ActiveSync from managed code, you need P/Invoke wrappers to access the Win32 ActiveSync libraries. We built two sets of wrappers – one in C# and the other in VB – that you can find on the CD accompanying this book. We used the P/Invoke Wizard introduced in Chapter 4, and then fine-tuned the output by hand. You could, of course, use the wizard to create your own P/Invoke Wrappers, since the Basic Edition is on the book CD. The Advanced Edition is available for order from The Paul Yao Company (details at <http://www.paulyao.com>). [\[Comment 21cs.61\]](#)

### Remote API Startup and Shutdown

Like other Win32 API function sets, RAPI has startup and shutdown functions. Before calling any other RAPI function, you must call a startup function: either `CeRapiInit` or `CeRapiInitEx`. The difference is that the first is a blocking function and the second is not. [\[Comment 21cs.62\]](#)

When startup is a success, your program does its work and then calls the shutdown function, `CeRapiUninit`. If you call a startup function, you must always call the shutdown function. This is true even if the startup fails. When we have run programs that did not follow this guideline, we notice that ActiveSync itself stops running properly until the offending program terminates. [\[Comment 21cs.63\]](#)

*CeRapiInit versus CeRapiInitEx*

At first glance, the `CeRapiInit` function seems the simplest way to establish a RAPI connection. But this is a blocking function, so a call at the wrong time causes a thread to hang, keeping a program in memory when it was supposed to have ended. By contrast, `CeRapiInitEx` is a non-blocking function. [\[Comment 21cs.64\]](#)

`CeRapiInitEx` notifies the caller that a connection has been established through a Win32 event (you can wrap a .NET event object around a Win32 event – see Chapter 14). This function accepts a pointer to a `RAPIINIT` structure for input, and it returns an `HRESULT` result code. Here are the C# declarations for `CeRapiInitEx` and the required `RAPIINIT` structure: [\[Comment 21cs.65\]](#)

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern int CeRapiInitEx (ref RAPIINIT pRapiInit);

[StructLayout(LayoutKind.Sequential, Pack=4)]
public struct RAPIINIT
{
    public int cbSize;
    public IntPtr heRapiInit;
    public int hrRapiInit;
};
```

PRODUCTION NOTE: Insert Win32 Icon Here.

**TIP Handling Win32 HRESULT Values**

Treat Win32 `HRESULT` values as `int` values, to stay compatible with the Common Language Specification. Avoid an overflow in C# with the unchecked keyword as shown here: [\[Comment 21cs.66\]](#)

```
int E_FAIL = unchecked((int)0x80004005);
```

To call the `CeRapiInitEx` function, first allocate a `RAPPINIT` structure and set `cbSize` to the structure size. If you set the incorrect size, `CeRapiInitEx` returns an error (like many Win32 functions). On return from the function call, the other two structure members hold return values. A Win32 event handle is returned in `heRapiInit`. After that event becomes signaled, `hrRapiInit` holds the initialization result code. [\[Comment 21cs.67\]](#)

*Shutdown with CeRapiUninit*

After any call to a RAPI initialization function – whether or not a connection was established – you must call `CeRapiUninit`. The C# P/Invoke wrapper for this function is shown here: [\[Comment 21cs.68\]](#)

```
[DllImport("rapi.dll")]
public static extern int CeRapiUninit();
```

*Two Approaches To Startup*

So far, we have looked at two functions and one structure that provide the first two functions you will need for all RAPI code that you write. Now it's time to look at these elements in the context of real working code. We are going to show two different approaches to handling RAPI startup. The first is short and self-contained, suitable to show all the startup details in one place. The second is a more "real-world" approach that involves creating a worker thread to handle the startup activity. [\[Comment 21cs.69\]](#)

**Method 1: Simple, Single-Threaded RAPI Startup**

Listing 14-1 shows the simplest possible RAPI startup and shutdown. This program includes all needed P/Invoke declarations for the two required rapi.dll functions:

CeRapiInitEx and CeRapiUninit. This program, located on the book CD, provides a quick starting point for experimenting with other RAPI functions. [\[Comment 21cs.70\]](#)

**Listing 14-1 RapiStartup.cs** [\[Comment 21cs.71\]](#)

```
// RapiStartup.cs - Simple thread-free RAPI startup
// with all P/Invoke declarations included.
//
// Code from _Programming the .NET Compact Framework with C#_
// and _Programming the .NET Compact Framework with VB_
// (c) Copyright 2002-2003 Paul Yao and David Durant.
// All rights reserved.

using System;
using System.Threading;
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace RapiStartup
{
    public class RapiStartup
    {
        const string m_strAppName = "RapiStartup";

        public RapiStartup()
        {
        }

        // -----
        // RAPI.DLL Definitions
        // -----
        public struct RAPIINIT
        {
            public int cbSize;
            public IntPtr heRapiInit;
            public int hrRapiInit;
        };

        [DllImport("rapi.DLL", CharSet=CharSet.Unicode)]
        public static extern int CeRapiInitEx (ref RAPIINIT p);
        [DllImport("rapi.DLL", CharSet=CharSet.Unicode)]
        public static extern int CeRapiUninit ();

        public const int S_OK = 0;

        // -----
        // Main -- Program entry point
        // -----
        public static void Main()
        {
            // Allocate structure for call to CeRapiInitEx
            RAPIINIT ri = new RAPIINIT();
            ri.cbSize = Marshal.SizeOf(ri);
        }
    }
}
```

```

        // Call init function
        int hr = CeRapiInitEx(ref ri);

        // Wrap event handle in corresponding .NET object
        ManualResetEvent mrev = new ManualResetEvent(false);
        mrev.Handle = ri.hrRapiInit;

        // Wait five seconds, then fail.
        if (mrev.WaitOne(5000, false) && ri.hrRapiInit == S_OK)
        {
            // Connection established.
            MessageBox.Show("Connection Established", m_strAppName);
        }
        else
        {
            // On failure, disconnect from RAPI.
            CeRapiUninit();

            MessageBox.Show("Timeout - No Device", m_strAppName);
            return;
        }

        // If we get here, we have established a RAPI connection

        // ToDo: Put your RAPI calls here...

        // Cleanup.
        CeRapiUninit();
    }

} // class RapiStartup
} // namespace RapiStartup

```

### Method 2: Multi-Threaded RAPI Startup

Our second method for starting RAPI involves a background thread. At the same time, this sample shows how to connect to the ActiveSync wrapper library. We have two such libraries – a C# version and a VB version – both of which are named `YaoDurant.Win32.Rapi.dll`. There are at least two ways to use these libraries: copy the source code declarations directly to your source code, or call into one of these libraries from your code. [\[Comment 21cs.72\]](#)

Three pieces are needed to allow an application to access a dynamic link library – no matter what language or API you are using: (1) a piece for the compiler, (2) a piece for the linker, and (3) a piece for the loader. Win32 programmers recognize these pieces as the (1) include (.h) file for the compiler, (2) the library (.lib) file for the linker, and (3) the dynamic link library itself for the loader. Here is how to provide these three pieces for a .NET DLL: [\[Comment 21cs.73\]](#)

- 1 For the compiler, add a `using` statement to your source file:
 

```
using YaoDurant.Win32;
```
- 2 For the linker, add a reference to the library in Visual Studio .NET:
- 3 Open the Solution Explorer (from the menu **View->Solution Explorer**).
- 4 Right click on the **References** entry and select **Add Reference...**
- 5 Click on the **Browse...** button, and

- 6 When you find the needed library click the **Open** button.
- 7 For the loader, copy the DLL to the directory of the program that needs to use it.

Once these steps are completed, you can access the RAPI types using the wrapper class name, `Rapi`. In a Visual Studio .NET editor window, when you type "Rapi", IntelliSense shows the types defined within the `Rapi` class. [\[Comment 21cs.74\]](#)

Listing 14-2 shows `StartupThread.cs`, a file taken from the `FindProgram` sample. It performs RAPI initialization on a background worker thread, which is more typical of what a production RAPI program would do. [\[Comment 21cs.75\]](#)

**Listing 14-2 RapiStartupThread.cs** [\[Comment 21cs.76\]](#)

```
// Program Name: FindProgram.exe
//
// File Name: RapiStartupThread.cs - Creates a background thread
// for the purpose of starting RAPI.
//
// Code from _Programming the .NET Compact Framework with C#_
// and _Programming the .NET Compact Framework with VB_
// (c) Copyright 2002-2003 Paul Yao and David Durant.
// All rights reserved.

using System;
using System.Threading;
using System.Windows.Forms;
using System.Runtime.InteropServices;
using System.Diagnostics;
using YaoDurant.Win32;

namespace FindPrograms
{
    // Table of reasons that WorkerThread calls into the
    // user-interface thread.
    public enum INVOKE_STARTUP
    {
        STARTUP_SUCCESS,
        STARTUP_FAILED,
        STATUS_MESSAGE
    }

    /// <summary>
    /// StartupThread - Wrapper class that spins a thread
    /// to initialize RAPI. Calls a delegate to report status.
    /// </summary>
    public class StartupThread
    {
        public string strBuffer;           // Inter-thread buffer
        public INVOKE_STARTUP itReason;    // Inter-thread reason

        private Thread m_thrd = null;     // The contained thread
        private Control m_ctlInvokeTarget; // Inter-thread control
        private EventHandler m_deleCallback; // Inter-thread delegate
        private bool m_bContinue; // Continue flag.
    }
}
```

```

public bool bThreadContinue // Continue property.
{
    get { return m_bContinue; }
    set { m_bContinue = value; }
}

/// <summary>
/// StartupThread - Constructor.
/// </summary>
/// <param name="ctl">Owner control</param>
/// <param name="dele">Delegate to invoke</param>
public StartupThread(Control ctl, EventHandler dele)
{
    bThreadContinue = true;
    m_ctlInvokeTarget = ctl; // Who to call.
    m_deleCallback = dele; // How to call.
}

/// <summary>
/// Run - Init function for startup thread.
/// </summary>
/// <returns></returns>
public bool Run()
{
    ThreadStart ts = null;
    ts = new ThreadStart(ThreadMainStartup);
    if (ts == null)
        return false;

    m_thrd = new Thread(ts);
    m_thrd.Start();
    return true;
}

/// <summary>
/// ThreadMainStartup - Start RAPI connection.
/// </summary>
private void ThreadMainStartup()
{
    // Allocate structure for call to CeRapiInitEx
    Rapi.RAPIINIT ri = new Rapi.RAPIINIT();
    ri.cbSize = Marshal.SizeOf(ri);

    // Call init function
    int hr = Rapi.CeRapiInitEx(ref ri);

    // Wrap event handle in corresponding .NET object
    ManualResetEvent mrev = new ManualResetEvent(false);
    mrev.Handle = ri.hrRapiInit;

    // Wait five seconds, then fail.
    if (mrev.WaitOne(5000, false) && ri.hrRapiInit == Rapi.S_OK)
    {
        // Notify caller that connection established.
        itReason = INVOKE_STARTUP.STARTUP_SUCCESS;
        m_ctlInvokeTarget.Invoke(m_deleCallback);
    }
}

```

```
        else
        {
            // On failure, disconnect from RAPI.
            Rapi.CeRapiUninit();

            strBuffer = "Timeout - no device present.";
            itReason = INVOKE_STARTUP.STATUS_MESSAGE;
            m_ctlInvokeTarget.Invoke(m_deleCallback);

            // Notify caller that connection failed.
            itReason = INVOKE_STARTUP.STARTUP_FAILED;
            m_ctlInvokeTarget.Invoke(m_deleCallback);
        }

        // Trigger that thread has ended.
        m_thrd = null;
    }
} // class StartupThread
} // namespace FindPrograms
```

The `StartupThread` class provides a wrapper around the startup thread. We put this class into its own file to make it easy for you to reuse. This class uses the `Invoke` function to communicate from the background thread to the user-interface thread. This class depends on having a control (or a form) and on that control having a delegate to receive the inter-thread calls. Our startup wrapper has 3 methods: a constructor, `Run`, and `ThreadMainStartup`. [\[Comment 21cs.77\]](#)

`ThreadMainStartup` is the thread entry point, and it does the actual work of calling `CeRapiInitEx` to initiate the RAPI connection, and calling `CeRapiUninit` if the attempt to connect fails. [\[Comment 21cs.78\]](#)

A background thread allows the user-interface thread to be available to receive user input. A downside to multi-threaded programming is that the two threads need to be carefully crafted to avoid conflict, which because they are often timing related are usually very hard to reproduce and so hard to find. Our approach to this problem is for each thread to run in a separate class and not interact with each other, with two exceptions: thread startup code, and thread-safe communication code. In this code, there is a one-way communication from the worker thread to the main thread. The communications is handled by the only function in the `Control` class guaranteed to be thread-safe:

`Control.Invoke`. [\[Comment 21cs.79\]](#)

#### **NOTE Why `Control.Invoke` is Thread Safe**

The `Control.Invoke` method is thread-safe – in both the Desktop Framework and the Compact Framework – because it relies on an underlying Win32 feature that is guaranteed thread-safe. That feature is the `SendMessage` function, a Win32 function that has been around since the first version of Windows. This function sends a Win32 message to a window, waiting for the message to be delivered before returning. If the recipient is busy with another message, `SendMessage` blocks until the previous message has been handled. Over the years, this function has been the backbone for inter-thread and inter-process communication in a variety of Windows technologies, including DDE, OLE, and COM. It is not surprising, then, that this time-tested mechanism is also used by the .NET control classes. [\[Comment 21cs.80\]](#)

The Desktop Framework provides two overloads for the `Invoke` method:

```
public object Invoke(Delegate);
public virtual object Invoke(Delegate, object[]);
```

Each overload accepts a parameter of type `Delegate` (similar to a C function pointer). The second overload accepts an array of values to be used as parameters to the called function, and is the version most often used in the Desktop Framework. The Compact Framework, however, only supports the first type of overload, and because this is a Compact Framework book we are going to resist the temptation to do things the Desktop Framework way. Here is a call to `Invoke`, to notify the main thread that a successful RAPI connection has been established: [\[Comment 21cs.81\]](#)

```
// Notify caller that connection established.
itReason = INVOKE_STARTUP.STARTUP_SUCCESS;
ctlInvokeTarget.Invoke(deleCallback);
```

The recipient of this function call is a function named `StartupCallback`, a member of the `MainForm` class in our `FindPrograms` sample. We cannot just pass a function pointer like we do in unmanaged C, however. Instead, we pass the .NET equivalent, a delegate object of type `EventHandler`. Here is the declaration for the delegate, along with the other declarations needed to support the startup thread: [\[Comment 21cs.82\]](#)

```
// Startup thread definitions
StartupThread thrdStartup = null;
private EventHandler deleStartup;
private bool bRapiConnected = false;
```

We initialize the delegate as follows: [\[Comment 21cs.83\]](#)

```
deleStartup = new EventHandler(this.StartupCallback);
```

and then pass it to the constructor for our startup thread wrapper object: [\[Comment 21cs.84\]](#)

```
// Create thread to connect to RAPI.
thrdStartup = new StartupThread(this, deleStartup);
if (!thrdStartup.Run())
    thrdStartup = null;
```

After creating the `StartupThread` object, we call the `Run` member to start it. This two-step approach – create the object, then initialize the object – may be familiar to C++ programmers from other object-oriented APIs. A two step approach works because a constructor cannot easily provide a return value, nor can it report a reason when it fails. A stand-alone initialization function – `Run` in this example – can do both. [\[Comment 21cs.85\]](#)

Listing 14-3 shows our inter-thread delegate function, `StartupCallback`. This function takes two parameters: `sender` and `e`. The parameters have names that suggest they might be useful, but neither in fact are accurate. The first, `sender`, identifies the

*recipient* of the function call – the control that is receiving the `Invoke` call – and not the sender as the name implies [\[Comment 21cs.86\]](#)

Since neither parameter is useful, we structure our code so that this function has only one possible caller, the thread wrapper object, `thrdStartup`. The recipient of the call then accesses that caller's public data to receive any parameters needed for the inter-thread function call, and respond accordingly. Because of the way `Invoke` works, our background thread is blocked while the main thread runs the `Invoke` target function. In this way, our code is thread-safe. [\[Comment 21cs.87\]](#)

**Listing 14-3 StartupCallback accepts Invoke calls from the startup thread** [\[Comment 21cs.88\]](#)

```
private void
StartupCallback(object sender, System.EventArgs e)
{
    INVOKE_STARTUP it = this.m_thrdStartup.itReason;
    switch(it)
    {
        case INVOKE_STARTUP.STARTUP_SUCCESS:
            m_bRapiConnected = true;
            EnableUI();
            break;
        case INVOKE_STARTUP.STARTUP_FAILED:
            ResetUI();
            break;
        case INVOKE_STARTUP.STATUS_MESSAGE:
            sbarMain.Text = m_thrdStartup.strBuffer;
            break;
    }
}
```

A key element connecting the main thread to the startup thread is the connection status flag, `m_bRapiConnected`. When the startup thread establishes the RAPI connection, it sends the main thread a `STARTUP_SUCCESS` code, and the main thread sets the connected flag to `true`: [\[Comment 21cs.89\]](#)

```
m_bRapiConnected = true;
```

This flag reminds us to shutdown our RAPI connection at the appropriate time. For example, as shown in the following code, when the main form closes we also shutdown any running worker threads: [\[Comment 21cs.90\]](#)

```
private void
MainForm_Closed(object sender, System.EventArgs e)
{
    // If threads are running, trigger shutdown.
    if (this.m_thrdStartup != null)
        this.m_thrdStartup.bThreadContinue = false;
    if (this.m_thrdFindFiles != null)
        this.m_thrdFindFiles.bThreadContinue = false;

    if (m_bRapiConnected)
    {
```

```
Rapi.CeRapiUninit();  
    m_bRapiConnected = false;  
}  
}
```

### *Defensive Coding*

RAPI is a communications protocol, which means you must code defensively. When a connection has been established, your program must be able to handle lengthy operations – reading or writing data – that may take a long time because either the connection is slow (USB or serial), or the volume of data is large, or both. Also, when communicating between two machines, the connection may get interrupted. `ActiveSync`, for example, is interrupted when a user removes a device from its docking cradle. In addition to interrupted connections, your code must handle the possibility that no connection – meaning no device – is available. [\[Comment 21cs.91\]](#)

Lengthy operations can be caused by blocking functions, by a slow connection, or simply because a lot of data must be moved. Blocking functions only return when the requested action is finished. For example, `CeRapiInit` returns only after a RAPI connection is established, and the function hangs when no device is present. Most RAPI functions, in fact, are blocking – a.k.a. "synchronous" – and finish their work before returning. Functions that return a large amount of data might take several seconds or even several minutes to finish. [\[Comment 21cs.92\]](#)

While these issues require some effort to address, none is entirely daunting. To prevent a blocking functions from hanging a program's user-interface, we recommend adding threads to your programs. As discussed in Chapter 14 (*Threads, Timers, & Notifications*), such programs have a dedicated user-interface thread, and then one or more worker threads that focus on a single task. Without threads, an application can become unresponsive to a user's input and perhaps make a user worry that the program has hung. Threads – and feedback to the user with progress information – helps ease such user concerns. [\[Comment 21cs.93\]](#)

The final challenge – a disconnected connection – can be a little more difficult. For individual function calls, you must use vigilance in checking return values. In addition, you will want to use exception handling (the `try` & `catch` keywords – see discussion in chapter 2) so that when something fails deep within your code, you can escape the problem by raising an exception and recovering from the failure higher up the call stack. Exception handling simplifies the handling of unexpected failures. [\[Comment 21cs.94\]](#)

---

### *Accessing the Object Store*

The majority of RAPI functions involve reading from or writing to the object store. As detailed in Chapter 17, the *object store* is the RAM-based storage area with three parts: a file system, the registry, and property databases. [\[Comment 21cs.95\]](#)

The Windows CE file system is very similar to the desktop file system: it is hierarchical, supporting nested subdirectories (sometimes called "folders"), and long file names – and long file paths – that can be up to 260 characters long. The file system in the object store provides the primary storage area for a Windows CE device. Additional file systems can be added – known as "installable file systems" – to extend available storage space. Two common installable file systems are Compact Flash memory cards and Secure Digital storage cards. Both are sometimes referred to as "memory cards" because they contain non-volatile flash memory, but both are used for file storage and

---

## *Thank You*

Thank you for taking the time to read this preview chapter. We hope it has provided you insights and tips to help with your Compact Framework programming project. You can help us create a better book by clicking the comment link at the end of each paragraph, and sending your comments and suggestions on our review web site.

---

## *Preview Chapter Text*

Our public review site provides the complete table of contents for the Compact Framework book at this link: <http://www.paul Yao.com/cfbook.htm>. That table of contents contains links to all the preview chapters. The preview chapter provides the complete outline of topics covered in a chapter, and also the first section or two from each chapter.

---

## *Complete Chapter Text*

You can get the complete text for each chapter, available to readers who register at our web site. Registration is simply and easy – we only ask for an email address. To register, click on this link: <http://www.paul Yao.com/ReaderFeedback/Logon.aspx>.

When you register, you can download the available chapters from the full-text Table of Contents, available at this link: <http://www.paul Yao.com/ReaderFeedback/default.aspx>. We notify registered readers of new chapters – and chapter updates – as they become available.