C/C++ Sync Suite Companion

Palm OS® Conduit Development Kit for Windows, Version 6.0.1

Written by Brent Gossett.
Technical assistance from Cole Goeppinger, Gerard Pallipuram, Thomas Butler, and Robert Rhode.

C/C++ Sync Suite Companion
Document Number 3013-006
May 14, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

Table of Contents

About This Document

The C/C++ Sync Suite is a component of the Palm OS® Conduit Development Kit (CDK) for Windows from PalmSource, Inc. It provides APIs, the C++ Generic Conduit Framework, samples, documents, and utilities to help developers create C API-based conduits that run on Windows computers. Key to the success of the Palm OS platform, conduits are software objects that exchange and synchronize data between an application running on a desktop computer and a Palm Powered™ handheld.

The C/C++ Sync Suite Companion provides an overview of how C API-based conduits operate and how to develop them with the C/C++ Sync Suite.

The sections in this introduction are:

Related Documentation

The latest versions of the documents described in this section can be found at

http://www.palmos.com/dev/support/docs/

The following documents are part of the CDK:

Document	Description
Introduction to Conduit Development	An introduction to conduits on the Windows platform. It describes how they relate to other aspects of the Palm OS platform, how they communicate with the HotSync® Manager, and how to choose an approach to conduit development. Recommended reading for developers new to conduits.
C/C++ Sync Suite Companion	An overview of how C API-based conduits operate and how to develop them with the C/C++ Sync Suite.
C/C++ Sync Suite Reference	A C API reference that contains descriptions of all conduit function calls and important data structures used to develop conduits with the C/C++ Sync Suite.
COM Sync Suite Companion	An overview of how COM-based conduits operate and how to develop them with the COM Sync Suite.
COM Sync Suite Reference	A reference for the COM Sync Suite object hierarchy, detailing each object, method, and property.
Conduit Development Utilities Guide	A guide to the CDK utilities that help developers create and debug conduits for Windows.

What this Document Contains

This section provides an overview of the chapters in this document:

- Chapter 1, "Introduction to the C/C++ Sync Suite." Helps you get started developing a conduit with the C/C++ Sync Suite. It introduces the concept of a conduit, what you need to develop a conduit, and what the C/C++ Sync Suite provides.
- Chapter 2, "Introducing C API-based Conduits." Introduces C API-based conduits.
- Chapter 3, "Implementing Conduit Entry Points." Describes the required and optional entry points that your conduit must implement for HotSync Manager to run your conduit.
- Chapter 4, "Using the Classic Sync Manager API." Provides an overview of the classic Sync Manager API and how to use it.
- Chapter 5, "Using the Extended Sync Manager API." Compares the extended and classic Sync Manger APIs.
- Chapter 6, "Using Expansion Technology." Describes how to work with handheld expansion cards and add-on devices from the desktop using the Expansion and Virtual File System (VFS) Managers.
- Chapter 7, "Debugging Conduits." Provides an overview of techniques that you can use to help debug your conduits.
- Chapter 8, "Writing a Desktop Notifier." Describes how to notifiy your desktop application when a HotSync operation starts and stops.
- Chapter 9, "Writing an Installer." Provides an overview of making it easy for users to install your conduits on a Windows-based desktop computer.
- Appendix A, "Quick Start: Using Visual C++ .NET to Build a Conduit." Guides you through the steps necessary to build and debug a sample conduit that uses the Generic Conduit Framework.

Changes to This Document

This section describes significant changes made in each version of this document. For a description of what's new in each version of the CDK, see Chapter 1, "What's New in the Palm OS CDK," on page 1 in *Introduction to Conduit Development*.

- Document 3013-006 for CDK 6.0.1
- Document 3013-005 for CDK 6.0
- Document 3013-004 for CDK 6.0

Document 3013-006 for CDK 6.0.1

The significant corrections and additions in this document version are listed by chapter below:

- Removed the "Using File Linking" chapter because the file link feature has been removed from HotSync Manager 6.0.1.
- In Chapter 3, "Implementing Conduit Entry Points," on page 17, removed section on file linking entry points.
- Added Appendix A, "Quick Start: Using Visual C++ .NET to Build a Conduit."
- Moved information about the HotSync log to "Adding Messages to the HotSync Log" on page 46 in *Introduction to Conduit Development*.

Document 3013-005 for CDK 6.0

The significant changes are listed by chapter below:

- Chapter 3, "Implementing Conduit Entry Points."
  - Noted that HotSync Manager versions 6.0 and later do not query a conduit for its MFC version.
  - Added to `GetConduitInfo()` sample in Listing 3.1 on page 21 cases for the new enum values that HotSync Manager 6.0 and later can pass in the *`infoType`* parameter.
- Chapter 5, "Using the Extended Sync Manager API."
  - Clarified that classic, extended, and schema databases exist in disjoint namespaces.

  - Added that record deletion and creation is supported while iterating.

- Chapter 6, "Using Expansion Technology."
  - Added that the import and export functions do not support schema databases.

Document 3013-004 for CDK 6.0

Most of the changes in this version document the new capabilities added in Sync Manager version 2.4 (HotSync Manager 6.0). These and other changes are listed below in chapter order:

- Chapter 2, "Introducing C API-based Conduits."
  - Updated "Registering C API-based Conduits with HotSync Manager" on page 10 to include folder-based registration.
- Chapter 3, "Implementing Conduit Entry Points."
  - Moved information about logging and notifiers to separate chapters.
- Chapter 4, "Logging Messages in the HotSync Log."
  - Split out as a separate chapter and added details on how the HotSync log works.
- Chapter 5, "Using the Extended Sync Manager API."
  - Added chapter to describe the differences between extended and classic Sync Manager APIs.
- Chapter 8, "Writing a Desktop Notifier."
  - Added chapter and an example notifier.

Additional Resources

- Documentation

  PalmSource, Inc. publishes its latest versions of this and other documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/

- Training

  PalmSource and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

  http://www.palmos.com/dev/support/kb/

- CDK Feedback

  Use this email address to provide feedback on the CDK: features you would like to see, bug reports, errors in documentation, and requests for Knowledge Base articles.

  cdk-feedback@palmsource.com

Conventions Used in this Document

This guide uses the following typographical conventions:

This style...	Is used for...
`sample`	Literal text such as filenames, commands, code elements such as functions, structures, and so on.
sample	Emphasis or to indicate a variable.
sample	Definition or first usage of a term, menu and menu item names, user-supplied text, window names in UI descriptions.
sample	Hypertext links.

1

Introduction to the C/C++ Sync Suite

This chapter introduces the C/C++ Sync Suite, including the concept of a conduit, what you need to develop a conduit, and what the C/C++ Sync Suite provides.

The sections in this chapter are:

NOTE: For more background on conduits and the different languages you can develop them in, see the *Introduction to Conduit Development*.

Definition of a Conduit

A **conduit** is a plug-in module for the HotSync® Manager application that synchronizes data for an application on a Palm Powered™ handheld with data for a desktop application or in a networked database, converting data formats between the two as necessary. HotSync Manager runs the conduits that have been installed on the user's desktop computer when the user places a handheld into the cradle and presses the HotSync button.[1] Conduits are typically designed to synchronize or back up data from a specific handheld application database. You can develop conduits with any of the suites that make up the Palm OS® Conduit Development Kit (CDK) for Windows from PalmSource, Inc. (See Chapter 10, "Conduit Development," on page 157 in the *Introduction*

1. HotSync Manager also supports connections via infrared, modem, and local area network.

to Conduit Development for a comparison of the suites provided in the CDKs.)

A **C API-based conduit** is a Windows DLL created with the C/C++ Sync Suite. Each C API-based conduit running on a Windows system is a standard dynamically linked library (DLL). Your conduit code can perform any actions that any other DLL can perform. However, to maintain the integrity of the HotSync process, conduits need to meet certain design goals, as described in "Conduit Design Philosophy" on page 43 in the *Introduction to Conduit Development*.

NOTE: You must build your C API-based conduit as a regular DLL. Do not build it as an extension DLL.

Any conduit developed with the C/C++ Sync Suite can be used to connect to any Palm Powered handhelds, including those developed by Palm OS platform licensees.

Conduits exchange and synchronize data between desktop applications or other system applications running on a Windows desktop computer and an application running on a Palm Powered handheld. To do so, the user must perform the following steps:

1. Install the Palm OS® Desktop software or other desktop application from a Palm OS licensee. This software includes HotSync Manager.
2. Install the cradle as described by the installation documentation provided with the handheld.
3. Insert the handheld into its cradle (or via another connection type).
4. Press the HotSync button on the cradle (or start the HotSync client application on the handheld and tap the HotSync button on the display).

HotSync Manager synchronizes each application by running conduits designed for that application. HotSync Manager can execute many conduits in the same synchronization session even if the conduits are written in different languages—for example, C/C++ and Java.

Many conduits (including the conduits for the four most popular applications built into Palm Powered handhelds) synchronize data

between the handheld and the desktop computer such that they are mirror images after synchronization. Other conduits perform more complex operations. The complexity of your conduit's behavior determines the development effort involved.

C/C++ Sync Suite Contents

This section describes the contents of the C/C++ Sync Suite in the following sections:

Overview

The C/C++ Sync Suite is part of the Palm OS Conduit Development Kit (CDK) for Windows from PalmSource, Inc. The C/C++ Sync Suite provides the following files, documentation, libraries, and sample code that you need to develop C API-based conduits and desktop applications that access handheld user data:

- **Documentation** – Includes the *Introduction to Conduit Development*, *Conduit Development Utilities Guide*, *C/C++ Sync Suite Companion*, and the *C/C++ Sync Suite Reference*. See "Related Documentation" on page x.
- **Library and Header Files for C APIs** — The libraries and header files you need to compile projects you develop with the C/C++ Sync Suite.
- **Palm Foundation Classes** — The desktop classes used by many samples and APIs provided in the C/C++ Sync Suite.
- **Generic Conduit Frameworks** — The base classes you can reference while building Generic Conduits in C++. Two frameworks are provided: one for synchronizing **classic database**s and one for **extended database**s.
- **XML Conduit Sample Code** — A sample based on the Generic Conduit Framework, which synchronizes an extended database with an XML file on the desktop.

- **Conduit Development Utilities** — Several indispensable tools to help you develop and debug conduits. These include the Conduit Wizard to quickly create a functional conduit and the Conduit Configuration utility to register your conduit with HotSync Manager.

Top-Level Directories

During installation, the following directory structure is created on your system for the C/C++ Sync Suite:

`<CDK>\Common\Bin\`
: Contains HotSync Manager, support DLLs, and conduit development utilities.

`<CDK>\Common\Docs\`
: Contains all CDK documents in HTML Help format. Note that the documentation is also integrated with Visual Studio .NET.

`<CDK>\C++\Common`
: Common source and header files for the extended Generic Conduit Framework.

`<CDK>\C++\Win`
: Source, header, libraries, and sample files that are specific to Windows.

Samples Directory

To make getting started easier, the C/C++ Sync Suite includes several samples that you can customize to suit your needs. All of the following directories are in `<CDK>\C++\Win\Samples`.

`GenericConduit\`

`Base\`
: The base Generic Conduit Framework for extended databases.

`Classic\`
: The Generic Conduit Framework for classic databases. Note that the `PFC_OLD` subfolder contains PFC source and header files that are specific to classic databases.

`SamplePrc\`
: A ported version of the Memo Pad application that uses an extended database rather than a classic database. You can use this application to experiment with the extended Generic Conduit Framework. Subfolders contain versions for use on a Palm OS Cobalt handheld and on Palm OS Cobalt Simulator.

`XML\`
: A subclassed version of the extended Generic Conduit Framework that synchronizes with an XML file on the desktop.

`MSVC.NET 7.0 Wizard\`
: The Conduit Wizard files for VC .NET 2002.

`MSVC.NET 7.1 Wizard\`
: The Conduit Wizard files for VC .NET 2003.

  If the CDK installer detects that VC .NET is installed, it installs the correct version of the Conduit Wizard automatically. It puts in the `Samples` folder only the version of the Conduit Wizard that it did not install. See "Installing the Conduit Wizard" on page 4 in *Conduit Development Utilities Guide*.

`PDNotify\`
: C source code for the notifier that communicates with the Palm OS® Desktop software via HotSync Manager using the Desktop Application Notifier API.

`UserDataApp\`
: C++ source code for a small application that calls the User Data API to view and modify desktop user information. A prebuilt executable is also provided in the `<CDK>\Common\Bin` folder.

2

Introducing C API-based Conduits

This chapter introduces C API-based conduits. When you design a conduit, it is essential that you understand how your synchronization process works and which part of the system is responsible for each corresponding synchronization action. Therefore, if you are new to conduit development, read the *Introduction to Conduit Development* before proceeding.

The sections in this chapter are:

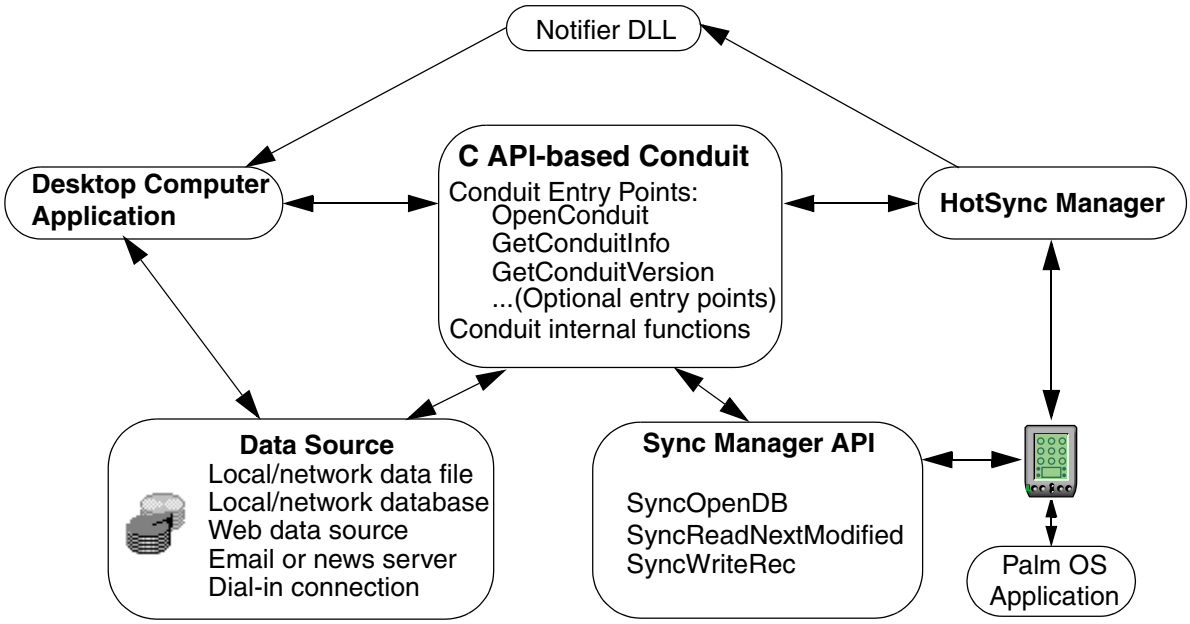
How the C/C++ Sync Suite Supports Conduits

This section provides an overview of how the C/C++ Sync Suite supports C API-based conduits:

Conduit Architecture

HotSync Manager communicates with a conduit by calling its several required entry points. A conduit accesses databases on the handheld by calling the Sync Manager API. Figure 2.1 shows the relationship between a conduit, HotSync Manager, the Sync Manager API, and the desktop application.

Figure 2.1 C/C++ Sync architecture



Conduit Entry Points

HotSync Manager calls several entry points in a conduit, including the `OpenConduit()` function, which allows the conduit to perform its tasks. A conduit must provide some of these entry points and can optionally provide others.

The easiest way to create a new conduit is to use the Conduit Wizard in Visual C++, as described in Chapter 2, "Conduit Wizard," on page 3 in the *Conduit Development Utilities Guide*. The Conduit Wizard is installed with the C/C++ Sync Suite for Windows.

Once you have used the wizard to create a functional conduit shell, you need to customize the conduit that has been generated. Modify the entry points described below:

- modify the `GetConduitInfo()` function to return information about your conduit
- modify `GetConduitVersion()` to return the version number of your conduit
- add code to the `OpenConduit()` function to perform your actual synchronization operations
- modify the `CfgConduit()`, `ConfigureConduit()`, or both functions to allow users to modify how your conduit behaves. Note that these functions are not required for a functional conduit, but PalmSource, Inc. strongly recommends including them.

For more information about the required and optional entry point functions, see Chapter 3, "Implementing Conduit Entry Points," on page 17.

Sync Manager API

The Sync Manager API is the low-level programmatic interface for communicating with the handheld. Conduits call Sync Manager functions to send data to and receive data from the handheld. If your conduit uses the Generic Conduit Framework, you may not need to use the Sync Manager API directly.

For more information about the Sync Manager API, see Chapter 4, "Using the Classic Sync Manager API," on page 29.

Generic Conduit Framework

Rather than relying directly on the Sync Manager API, you can develop your conduit using the C++ Generic Conduit Framework provided in the C/C++ Sync Suite. These C++ classes manage a conduit's tasks at a higher level and provide some base functionality for you; specifically, reconciliation of changes on both the handheld and desktop computer. The Conduit Wizard can create conduit class objects for you, and you then override certain methods to create the implementation of your conduit.

Registering C API-based Conduits with HotSync Manager

As with all types of conduits for Windows, you must register a C API-based conduit before HotSync Manager can run it. This topic is covered primarily in Chapter 6, "Registering Conduits and Notifiers with HotSync Manager," on page 73 in *Introduction to Conduit Development*.

Because you must register a conduit even during development and testing, before you have written an installer to deploy your conduit, you have two options for quickly and easily registering your conduit on your development system:

- Implement a folder-registered conduit. Registration is then simply a matter of copying your conduit it to a `Conduits` folder. This method is supported by HotSync Manager versions 6.0 and later. For details, see "Registering a Conduit by Folder" on page 112.
- Conventionally register your conduit using the Conduit Configuration (CondCfg) utility provided in the CDK. CondCfg is a Windows application that allows you to register your conduit using a graphical user interface on your development machine. CondCfg is in the `<CDK>\Common\Bin` directory. For more information, see Chapter 3, "Conduit Configuration Utility," on page 11 in the *Conduit Development Utilities Guide*.

IMPORTANT: CondCfg enables you to conventionally register your conduit during development and testing without writing code to call Conduit Manager. However, CondCfg is not an end-user utility, so PalmSource does not permit you to redistribute it with your conduit.

When you are ready to develop an installer to deploy your conduit on end-user machines, see Chapter 9, "Writing an Installer," on page 99 for details on calling the Conduit Manager API at install time to register a C API-based conduit.

Conduit Design Decisions

This section provides information that you can use to make decisions regarding the top-level design of your conduit.

IMPORTANT: If your design only needs to convert between handheld and desktop data formats, then you may be able to use the HotSync Exchange feature of HotSync Manager versions 6.0 and later; if so, you do not need to develop a conduit at all. See Chapter 3, "Using HotSync Exchange," on page 27 in the *Introduction to Conduit Development*.

If your conduit needs to do more than simple text-based synchronization, you can either decide to take advantage of the Generic Conduit Framework or write a conduit from scratch. To understand whether you may want to use the Generic Conduit Framework, first answer these questions:

- Will the conduit synchronize data such that both sides result in mirror-image data?

  If yes, you may want to use the Generic Conduit Framework because it implements logic to reconcile modifications made on both the desktop and handheld. You may need only to implement methods to read and write your desktop data format.

The most important considerations when designing your conduit are:

- how the data is stored on both platforms
- whether synchronization will be one-directional, mirror-image, or transaction-based

Your answers to the following questions will help determine the type of conduit you need to implement:

- Do the existing desktop computer databases offer a public application programming interface for record I/O? If so, is this API packaged as a stand-alone library?
- Does each record have a unique record ID field? If so, can the conduit perform keyed look-ups by Record ID? If so, can the conduit re-assign the record ID to a record? Is the record ID format on the desktop computer compatible with the record

ID format on the handheld? PalmSource, Inc. strongly recommends that record IDs get assigned only on the handheld.

- Does each record maintain a status flag field? If so, are the available status values Add, Modify, and Delete? How will you clear the status flags at the end of synchronization operations?
- Is there an easy way to detect modified records? A time stamp indicating the last modified date (with hours, minutes, and seconds) will work for this purpose. Your conduit needs this capability to support fast sync.
- Is the concept of categories supported in the existing databases? If so, what is the limit on number of categories allowed? Will category information be synchronized? How will the desktop categories be mapped to the categories on the handheld?
- Is the possibility of downloading only a subset of field data to the handheld acceptable? If so, consider how to retain that data on the desktop computer during the synchronization process.
- Will the conduit need to synchronize more than one database on the handheld to one or more database(s) on the desktop computer (multi-database synchronization)?
- Is one-way database copying desired?
- Will the conduit behave differently during a fast sync than during a slow sync?
- Is the data to be synchronized at the record level, so that each record that has changed is updated on both sides?
- If record-level synchronization is desired, do you want to use the synchronization logic implemented in the Generic Conduit Framework?
- Will the conduit need to synchronize one database on the handheld to one database on the desktop computer (single database synchronization)?
- How well do the desktop computer record fields and handheld records map to each other?
- Do your desktop and handheld applications support the concept of multiple databases or multiple users?

- Will archiving be supported?
- Can you pre-process data for efficiency, to keep the synchronization time to a minimum, and to avoid time-outs? For example, if your conduit accesses information over the Internet, it is best to access that information prior to beginning the synchronization.

Outline of Conduit Development

These five steps outline the process of creating a conduit with the CDK. For a more detailed description of how to set up your system and get started, see Appendix A, "Quick Start: Using Visual C++ .NET to Build a Conduit," on page 171.

A. Install Conduit Development Tools

You need to install the following software on your Windows desktop computer:

1. Install a development environment supported by the CDK. For a list, see Chapter 1, "What's New in the Palm OS CDK," on page 1.
2. Install the C/C++ Sync Suite, part of the CDK for Windows.

For more information about tools available to help you create and debug a conduit, see the *Conduit Development Utilities Guide*.

B. Generate a Sample Conduit to Use as a Shell

The easiest way to get started is to generate sample code included with the C/C++ Sync Suite. This code is for a fully functional and syntactically correct conduit that you can use as a shell.

In Visual C++ .NET:

1. Run the Conduit Wizard that is installed in VC++ .NET when you install the C/C++ Sync Suite. (The Conduit Wizard is available only in Visual C++ .NET.)
2. The first and most basic choice you need to make is which of the following you want the wizard to use when it generates your code:

- Generic Conduit Framework
- Conduit entry points only (no synchronization logic)

If you choose the Generic Conduit Framework, follow Appendix A, "Quick Start: Using Visual C++ .NET to Build a Conduit," on page 171. If instead you choose to write your own synchronization logic and use the lower-level Sync Manager API, use the wizard to generate entry points only, without any synchronization logic.

3. Use the wizard to generate a solution file and the code you specified.

The quick-start guide steps you through the process to create a conduit for the Memo Port application provided in the CDK.

For more information specifically about using the Conduit Wizard, see Chapter 2, "Conduit Wizard," on page 3 in the *Conduit Development Utilities Guide*.

C. Take a Trial Run

Test drive your sample conduit to verify that you can build and register it, and that the HotSync Manager application can run it:

1. Build the conduit as generated by the wizard without making any changes to the source files.
2. Register your conduit. If you implement a folder-registered conduit as done in the quick-start guide, then you need only to put your conduit in the `Conduits` folder—which the Conduit Wizard does automatically. However, if you are conventionally registering your conduit, then you can use the Conduit Configuration (`CondCfg.exe`) utility. (This utility is not intended for end users to use to install your conduit. You will still need to ship an installer as described in "E. Create an Install Program for Your Conduit" on page 16.)
3. Run your conduit by performing a standard HotSync operation. View the HotSync log to verify that your conduit ran.

This is where the quick-start guide ends. It helps you confirm that your system is set up correctly and that you can build and run a sample conduit.

IMPORTANT: Be sure that HotSync Manager can run the conduit you generated with the Conduit Wizard before you begin modifying any of the code. If it does not run, review the quick-start guide again to be sure you have built and registered the conduit correctly.

D. Implement the Conduit Entry Points

The sample code you generated with the Conduit Wizard provides complete implementations for the `GetConduitVersion()`, `GetConduitName()`, and `GetConduitInfo()` entry points. Change these only if you decide to change the name that you assigned to the conduit in the wizard.

You need to add code to the `OpenConduit()` entry point, which is where the interesting work of your conduit is done. The HotSync Manager application calls this entry point to have your conduit exchange data between the desktop computer and handheld. Most conduits perform the following basic steps:

1. Open the database on the handheld and open the data source on the desktop computer.
2. Use the Sync Manager API to read data from the handheld, and use the desktop application's native API to read data from the desktop data source. Sync Manager is the lowest-level way your conduit can communicate with the handheld. If you use the Generic Conduit Framework, the methods specified in these classes make the appropriate Sync Manager calls for you.
3. Exchange or synchronize the data as required.
4. Write the resulting data back to the handheld database and desktop data source.
5. Clear the handheld's record status flags.
6. Close the handheld database and the desktop data source.

For debugging and troubleshooting assistance, see Chapter 7, "Debugging Conduits," on page 85.

The Conduit Wizard generates a version of the `CfgConduit()` and `ConfigureConduit()` entry points, which display a default

configuration dialog box for your conduit. The default dialog box includes basic configuration options for the user: Synchronize the files, Desktop overwrites handheld, Handheld overwrites desktop, Do nothing. If you need to provide different options for users of your conduit, modify these entry points.

For more information about these and other, optional conduit entry points, see Chapter 3, "Implementing Conduit Entry Points," on page 17 in this document or Chapter 6, "Conduit Entry Point API," on page 495 in the *C/C++ Sync Suite Reference*.

E. Create an Install Program for Your Conduit

You need to make it easy for end users to install your conduit on their computers. The Conduit Configuration developer utility is not appropriate for users, so you must create an installer program.

You must also remember to test both the installation and removal (uninstallation) of your conduit on user systems, including verification that the HotSync Manager application continues to run properly after the user installs or uninstalls your conduit.

Your installer can make calls to the Conduit Manager, Install Aide, and User Data APIs to register your conduit, install your handheld application, and modify desktop user information. For more information about these APIs, see Chapter 9, "Writing an Installer," on page 99.

3

Implementing Conduit Entry Points

This chapter describes the required and optional entry points that your conduit must implement for HotSync® Manager to run your conduit. The sections in this chapter are:

Overview

There are two types of entry points that HotSync Manager calls in your conduit:

- Required Entry Points—must be implemented in each conduit.
- Optional Entry Points—enable users to configure your conduit via the HotSync Manager **Custom** dialog box.

This section provides a brief description of these conduit entry points. For more information about these functions, see Chapter 6, "Conduit Entry Point API," in the *C/C++ Sync Suite Reference*.

Table 3.1 provides a summary of the entry point functions.

Table 3.1 Conduit entry points called by HotSync Manager

Function Type	Function Name	Description
Required	`GetConduitVersion()`	Returns the conduit's version number. The supported range for conduit versions is `0x00000101` to `0x00000200`
Required	`OpenConduit()`	The main conduit entry point.
Required	`GetConduitInfo()`	Returns information about the conduit to HotSync Manager.
Customization (optional)	`ConfigureConduit()` (prior to HotSync 3.0.1) or `CfgConduit()` (HotSync 3.0.1 or later)	Presents a dialog that allows the user to configure the conduit. PalmSource, Inc. strongly recommends that you provide this function.
Optional	`GetConduitName()`	Returns the conduit's name for HotSync Manager to display. Required only for versions of HotSync Manager earlier than 3.0.1.

Required Entry Points

This section describes the following entry points for which you must implement your own versions:

- GetConduitInfo()
- GetConduitVersion()
- OpenConduit()

For details on these entry points, see Chapter 6, "Conduit Entry Point API," on page 495 the *C/C++ Sync Suite Reference*.

GetConduitInfo()

HotSync Manager calls the `GetConduitInfo()` function to retrieve information about your conduit. Your implementation of this function must respond to several different information requests, including requests for the following:

- the display name of your conduit
- your conduit's default action

NOTE: HotSync Manager versions 6.0 and later do not query a conduit for its MFC version, so responding to this query is optional.

However, a conduit intended to work with earlier versions of HotSync Manager must respond. In this case, you *must* implement `GetConduitInfo()` and return the appropriate MFC version constant. If you are recompiling a conduit you created with an older version of the CDK and did not originally implement `GetConduitInfo()` (now required), HotSync Manager assumes that your conduit is built on MFC 4.1. If it is not, HotSync Manager will crash when it calls your conduit.

HotSync Manager versions 6.0 and later request three additional, but optional, pieces of information:

- Registration information. Your conduit must respond if it is folder-registered (see "Registering a Conduit by Folder" on page 112), but not if it is a **conventionally registered conduit**.
- Whether your conduit's name should not be displayed in the HotSync Manager **Custom** and **HotSync Progress** dialog boxes.
- Whether HotSync Manager should always run your conduit, or only when an application with the same creator ID is on the handheld.

The Generic Conduit Framework's version of the `GetConduitInfo()` function, which is shown in Listing 3.1, is typical. Based on the *`infoType`* parameter passed in, it returns the requested type of information.

Listing 3.1 Example of `GetConduitInfo()` entry point

```
ExportFunc long GetConduitInfo(ConduitInfoEnum infoType, void *pInArgs,
   void *pOut, DWORD *pdwOutSize)
{
   // Do some error checking on passed-in parameters.
   // Do not modify these lines.
   if (!pOut)
      return CONDERR_INVALID_PTR;
   if (!pdwOutSize)
      return CONDERR_INVALID_OUTSIZE_PTR;

   // HotSync Manager requests information from your conduit through
   // repeated calls to GetConduitInfo.
   // The type of information requested is determined by infoType.
   // Your conduit responds appropriately.
   switch (infoType) {

      case eConduitName:
         // HotSync Manager wants to know the conduit name.
         // When HotSync Manager requests the conduit name, it also
         // passes in a ConduitRequestInfoType structure that
         // your conduit can examine, if necessary. The following
         // block of code shows how to check the version of this
         // structure in case PalmSource, Inc. changes the structure in the
         // future. You can omit this code if you do not need any
         // information in the ConduitRequestInfoType structure.
         if (!pInArgs)
            return CONDERR_INVALID_INARGS_PTR;
         ConduitRequestInfoType *pInfo;
         pInfo = (ConduitRequestInfoType *)pInArgs;
         if ((pInfo->dwVersion != CONDUITREQUESTINFO_VERSION_1) ||
            (pInfo->dwSize != SZ_CONDUITREQUESTINFO))
            return CONDERR_INVALID_INARGS_STRUCT;
         // OK, actually return the name of the conduit taken from the
         // resource IDS_CONDUIT_NAME.
      if (!::LoadString((HINSTANCE)hLangInstance, IDS_CONDUIT_NAME,
         (TCHAR*)pOut, *pdwOutSize))
         return CONDERR_CONDUIT_RESOURCE_FAILURE;
      break;

      case eDefaultAction:
         // HotSync Manager wants to know the default sync action.
         if (*pdwOutSize != sizeof(eSyncTypes))
            return CONDERR_INVALID_BUFFER_SIZE;
         (*(eSyncTypes*)pOut) = eDoNothing;
      break;
```

```
case eRegistrationInfo:
   // HotSync Manager wants your folder-based conduit's registration
   // information. Conventionally registered conduits do not respond.
   if (*pdwOutSize != sizeof(RegistrationInfoType)) {
      return CONDERR_INVALID_BUFFER_SIZE;
   }
   RegistrationInfoType* sRIT;
   sRIT = (RegistrationInfoType*) pOut;
   DWORD dwCreatorID;
   int iError;
   iError = CmConvertStringToCreatorID(GENERIC_CONDUIT_CREATOR_ID,
      &dwCreatorID);
   sRIT->dwCreatorID = dwCreatorID;
   sRIT->dwPriority = GENERIC_CONDUIT_PRIORITY;
   _tcscpy(sRIT->szLocalDirectory, GENERIC_CONDUIT_LOCAL_DIRECTORY);
   _tcscpy(sRIT->szLocalFile, GENERIC_CONDUIT_LOCAL_FILE);
   _tcscpy(sRIT->szRemoteDB, GENERIC_CONDUIT_REMOTE_DB);
   if (!::LoadString((HINSTANCE)hLangInstance, IDS_CONDUIT_NAME,
      sRIT->szTitle, sizeof(sRIT->szTitle))) {
      return CONDERR_CONDUIT_RESOURCE_FAILURE;
   }
break;

case eRunAlways:
   // HotSync Manager wants to know whether to run your conduit even
   // if there is no application on the device with the same creator ID.
   if (*pdwOutSize != sizeof(SInt8)) {
      return CONDERR_INVALID_BUFFER_SIZE;
   }
   // Run this conduit even if the creator ID it is registered for is
   // not on the device.
   *(SInt8*)pOut = 1;
break;

case eDoNotDisplayProgress:
   // HotSync Manager wants to know whether to display your conduit's
   // name in the HotSync Progress dialog box's Status field.
   if (*pdwOutSize != sizeof(SInt8)) {
      return CONDERR_INVALID_BUFFER_SIZE;
   }
   // Do not opt out of HotSync Progress dialog box.
   *(SInt8*)pOut = 0;
break;
```

```
      case eDoNotDisplayInConduitListForUser:
         // HotSync Manager wants to know whether to display your conduit's
         // name in HotSync Manager's Custom dialog box.
         if (!pInArgs) {
            return CONDERR_INVALID_INARGS_PTR;
         }
         if (*pdwOutSize != sizeof(SInt8)) {
            return CONDERR_INVALID_BUFFER_SIZE;
         }
         // ConduitRequestInfoType* pInfo = (ConduitRequestInfoType*) pInArgs;
         // Decide whether to display conduit name in Custom list for
         // particular pInfo->szUser
         *(SInt8*)pOut = 0;
      break;

      case eMfcVersion:
         // HotSync Manager wants to know the version of MFC.
         // Only HotSync Manager versions earlier than 6.0 request this
         // information. If you conduit works only with HotSync Manager 6.0 or
         // later, it does not need to respond.
         if (*pdwOutSize != sizeof(DWORD))
            return CONDERR_INVALID_BUFFER_SIZE;
         (*(DWORD*)pOut) = MFC_NOT_USED;
         break;

   default:
      return CONDERR_UNSUPPORTED_CONDUITINFO_ENUM;
   }
   return 0;
}
```

GetConduitVersion()

HotSync Manager calls the `GetConduitVersion()` function to retrieve the version of your conduit that is running on the desktop computer. Your implementation must pack your major version number into the high byte of the low word in the result, and must pack your minor version number into the low byte of the low word in the result.

The Generic Conduit Framework's version of the `GetConduitVersion()` function, which is shown in Listing 3.2 is typical. This function simply returns a constant value that specifies the version number.

Listing 3.2 Example of the `GetConduitVersion()` entry point

```
#define GENERIC_CONDUIT_VERSION 0x00000102

ExportFunc DWORD GetConduitVersion()
{
   return GENERIC_CONDUIT_VERSION;
}
```

OpenConduit()

HotSync Manager calls the `OpenConduit()` function to begin the process of synchronizing data between the desktop computer and the handheld. For conduits using the Generic Conduit Framework, your implementation of `OpenConduit()` needs to perform operations such as the following:

- instantiate an object of the `CSynchronizer` class and specify whether your conduit needs to handle categories or the application info block
- engage your synchronization by calling the `CSynchronizer`'s `Perform()` method
- delete the conduit's `CSynchronizer` object

Listing 3.3 shows the Generic Conduit's `OpenConduit()` entry point.

Listing 3.3 Example of the `OpenConduit()` entry point

```
ExportFunc long OpenConduit(PROGRESSFN pFn, CSyncProperties& rProps) {
   if (pFn) {
      CSynchronizer* pGeneric;

      // You will need to make this decision in your code.
      pGeneric = new CSynchronizer(rProps, GENERIC_FLAG_CATEGORY_SUPPORTED |
         GENERIC_FLAG_APPINFO_SUPPORTED);
      if (pGeneric) {
         retval = pGeneric->Perform();
         delete pGeneric;
      }
   }
   return(retval);
}
```

Optional Entry Points

This section describes the following optional entry points:

- CfgConduit()
- ConfigureConduit()
- GetConduitName()

You can implement these functions in your conduit, or you can use the default functionality, which is described in Chapter 6, "Conduit Entry Point API," on page 495 of the *C/C++ Sync Suite Reference*. PalmSource, Inc. recommends that you provide your own implementation of each of these functions.

CfgConduit()

HotSync Manager calls your `CfgConduit()` function when the user clicks **Custom** from the HotSync Manager menu, chooses your conduit, and clicks **Change**.

The `CfgConduit()` function is the HotSync Manager 3.0 and later version of the `ConfigureConduit()` function. The `CfgConduit()` function receives more information than the `ConfigureConduit()` function does, in the form of an extensible data structure.

HotSync Manager 3.0 and later attempts to call `CfgConduit()` first. If your conduit does not provide this entry point, HotSync Manager attempts to call the `ConfigureConduit()` function.Versions of HotSync Manager prior to version 3.0 unconditionally call the `ConfigureConduit()` function.

IMPORTANT: PalmSource, Inc. strongly recommends providing this entry point. Otherwise, nothing happens when the user chooses to customize your conduit. If your conduit does not support configuration, it is still a good idea to provide a version of this function that notifies the user of this limitation.

When HotSync Manager calls the `CfgConduit()` function, your conduit responds by displaying a configuration dialog box. This dialog box allows the user to specify what actions your conduit is to perform. Figure 3.1 shows the configuration dialog box displayed by the To Do List conduit.

Figure 3.1 "Custom" dialog box for the To Do List conduit

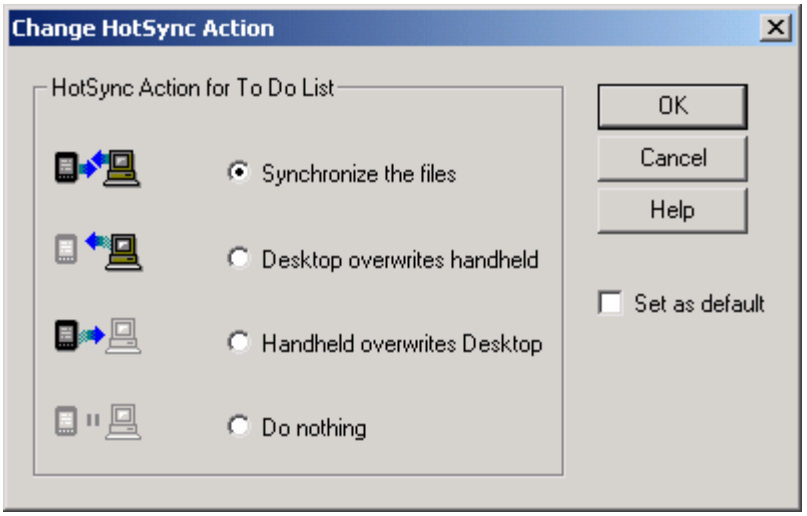


Figure 3.1 shows a typical configuration dialog box for a mirror-image synchronization. Your conduit might allow additional configuration options.

ConfigureConduit()

Versions of HotSync Manager earlier than version 3.0 call the `ConfigureConduit()` function instead of the `CfgConduit()` function. The purpose of each function is exactly the same.

GetConduitName()

HotSync Manager can call the `GetConduitName()` function to retrieve the name of your conduit. Your implementation fills in the name buffer with your conduit's name, and returns a status value that indicates whether the operation was successful.

HotSync Manager versions 3.0.1 and later attempt to retrieve the name of your conduit in this sequence:

1. Read the `Name` entry written with the CondCfg utility or the Conduit Manager API during conduit installation.
2. Else call `GetConduitInfo()`.
3. Else call `GetConduitName()`.

Therefore you are not required to implement `GetConduitName()` if your conduit needs to be compatible with only HotSync Manager versions 3.0.1 and later.

HotSync Manager versions 3.0 and earlier require that your conduit implement `GetConduitName()`. In which case, your implementation must fill in the `name` buffer with a string that you want HotSync Manager to display to the user.

The Generic Conduit Framework's version of the `GetConduitName()` function, which is shown in Listing 3.4, is typical. It passes back a pointer to the string defined in the string table (`IDS_CONDUIT_NAME`), and returns 0 to indicate success.

Listing 3.4 Example of the `GetConduitName()` entry point

```
ExportFunc long GetConduitName(char* pszName, WORD nLen)
{
   long retval = -1;

   if (::LoadString((HINSTANCE)hLangInstance, IDS_CONDUIT_NAME, pszName, nLen))
      retval = 0;

   return retval;
}
```

4

Using the Classic Sync Manager API

The classic Sync Manager API is a low-level programming interface to HotSync® operations with **classic database**s. This chapter provides an overview of the Sync Manager and how to use it in these sections:

The Sync Manager API libraries are located in the `<CDK>\C++\Win\lib` directory of the C/C++ Sync Suite.

Classic Sync Manager Overview

The classic Sync Manager manages all communications between the handheld and the desktop computer during synchronization operations with classic databases. This allows conduits to perform their operations without any consideration of how the data is moving between the handheld and desktop.

Your conduit does not need to distinguish between direct cable, modem, or network connections. In fact, your code does not need do anything with respect to the mechanics of exchanging data between the desktop computer and the handheld.

NOTE: The Sync Manager access only Palm OS databases on the handheld. To access cards in expansion slots, call the Expansion Manager and Virtual File System Manager described in Chapter 6, "Using Expansion Technology," on page 49.

HotSync Manager calls each conduit that is properly registered with it on the user's desktop computer. HotSync Manager calls an entry point, the `OpenConduit()` function, in your conduit, which allows your conduit to begin operation. The conduit can then call Sync Manager functions to exchange data with the handheld.

To use the Sync Manager, your conduit must first call the Sync Manager `SyncRegisterConduit()` function to configure the conduit. You can then call the Sync Manager functions to exchange data with the handheld. When you finish your synchronization process, call the `SyncUnRegisterConduit()` function to release the Sync Manager to other processes.

NOTE: The Sync Manager is not guaranteed to be thread-safe, which means that you must make all of your Sync Manager calls from the same thread that launched your conduit.

Sync Manager Error Checking

Because the connection to the handheld might be interrupted at any time, you must be sure to check for errors after calling any Sync Manager function.

IMPORTANT: If your call to a Sync Manager function returns an error, the other values returned by that function are considered undefined. Using these values can cause data corruption.

Classes and Structures in Sync Manager

The Sync Manager API includes a number of classes that are described in this chapter. All of these classes consist of data members only and can be used in the same manner as you use standard C data structures.

Sync Manager and Performance

You need to keep in mind that many of the calls you make to the Sync Manager functions require the transfer of data between your desktop computer and the handheld. These calls are input/output bound and may require significant amounts of time to complete. For best performance of your conduit, you need to structure your logic to minimize the number of Sync Manager function calls.

Specifically, you should design your code with the following performance-enhancing tips in mind:

- Avoid reading a record or information block more than once from the handheld database. You can cache information in the desktop computer to eliminate the need to access the same information more than once.
- When you are reading information from a database, allocate a buffer (up to 64 KB) that is large enough to store the largest possible record size in that database. This is significantly faster than making one call to determine the data size and another call to actually read the data.

The following Sync Manager functions are not I/O bound and can therefore be called without significantly impacting the performance of your conduits:

- `SyncGetAPIVersion()`
- `SyncHHToHostDWord()`
- `SyncHHToHostWord()`
- `SyncHostToHHDWord()`
- `SyncHostToHHWord()`
- `SyncYieldCycles()`

Typical Flow of Sync Manager Calls

The following sections describe the flow of calls from a typical conduit to the Sync Manager. These calls fall into the following categories:

- registering the conduit with the Sync Manager
- opening a database
- reading and writing records
- working with record categories
- closing the database
- unregistering the conduit with the Sync Manager

Registering Your Conduit with Sync Manager

You must register your conduit with the Sync Manager by calling the `SyncRegisterConduit()` function. The Sync Manager returns a handle for your conduit, which you use in subsequent calls. When your conduit is done, you unregister it by calling the `SyncUnRegisterConduit()` function. Most conduits make these calls in their `OpenConduit()` functions.

NOTE: Be careful not to confuse registering with the Sync Manger and registering with HotSync Manager. Your conduit must register with the *Sync* Manager at run time. You must register your conduit with *HotSync* Manager at install time, otherwise HotSync Manager does not run your conduit (see "Registering C API-based Conduits with HotSync Manager" on page 10).

Opening Your Database

You can use the `SyncCreateDB()` function to create a new, opened database on the handheld, or you can use the `SyncOpenDB()` function to open an existing handheld database. Listing 4.1 shows a section of code from an `OpenDataBase()` function. This function calls both the `SyncCreateDB()` and `SyncOpenDB()` functions.

Listing 4.1 Opening a handheld database

```
BOOL OpenDataBase( CLogMgr& theLog, const char * pName, DWORD dbCreator,
   DWORD dbType, BYTE& hDataBase, WORD& dbRecordCount,
   BOOL clearDB )
{
long retVal;
   // Open the remote database.
   retVal = SyncOpenDB( pName, 0, hDataBase );
   if ( retVal )
   {
      if ( retVal == SYNCERR_FILE_NOT_FOUND )
      {
         TRACE( "SyncOpenDB( %s ) failed [ %lX ], attempting to create DB.\n",
            pName, retVal );

         // The database does not exist, so create it.
         CDbCreateDB  createInfo;
         memset( &createInfo, 0, sizeof( CDbCreateDB ) ) ;

         createInfo.m_Creator   = dbCreator;
         createInfo.m_Type      = dbType;
         createInfo.m_Flags     = eRecord;          // eRecord;
         createInfo.m_CardNo    = 0;                // Target card number
         createInfo.m_Version   = 0;

         ASSERT( strlen( pName ) <= DB_NAMELEN );
         strcpy( createInfo.m_Name, pName );

         retVal = SyncCreateDB( createInfo );
         if ( retVal )
         {
            CString theDBName( pName );
            // fatal error
            theLog.Message( IDS_CANT_CREATE_DB, theDBName, TRUE );


            TRACE( "SyncCreateDB( %s ) failed [ %lX ].\n", pName, retVal );

            return FALSE;
         }
         else
         {
            // Database is open with handle createInfo.m_FileHandle.
            hDataBase = createInfo.m_FileHandle;

            TRACE( "SyncCreateDB( %s ) succeeded.\n", pName );
         }
      }
```

```
      else
      {
         CString theDBName( pName );
         // fatal error
         theLog.Message( IDS_CANT_OPEN_DB, theDBName, TRUE );

         TRACE( "SyncOpenDB( %s ) failed [ %lX ].\n", pName, retVal );

         return FALSE;
      }
   }
   // If clearDB, then clear all the existing records in the database.
   if ( clearDB )
   {
      retVal = SyncPurgeAllRecs( hDataBase );
      if ( retVal )
      {
         CString theDBName( pName );
         // fatal error
         theLog.Message( IDS_CANT_PURGE_DB, theDBName, TRUE );

         TRACE( "SyncPurgeAllRecs( %s ) failed [ %lX ].\n", pName, retVal );

         return FALSE;
      }
      else
      {
         TRACE( "SyncPurgeAllRecs( %s ) succeeded.\n", pName );
      }
   }

   // Determine how many records are in the remote database.
   retVal = SyncGetDBRecordCount( hDataBase, dbRecordCount );
   if ( retVal )
   {
      if ( hDataBase ) SyncCloseDB( hDataBase );

      CString theDBName( pName );
      // fatal error
      theLog.Message( IDS_CANT_GET_DBRECCOUNT, theDBName, TRUE );

      TRACE( "SyncGetDBRecordCount( %s ) failed [ %lX ].\n", pName, retVal );

      return FALSE;
   }

   return TRUE;

} // OpenDataBase
```

Working with Databases

Each database is a collection of records with the following characteristics:

- Each record in a database has an ID that is unique within the database.
- Each record in a database has flags that specify whether it has been marked as private, deleted, modified, archived, or busy.
- Each record in a database belongs to a category.
- The database contains an application information block that stores global information about the database. This block is typically used to store the names of categories for records in the database.
- The database has a sort information block that stores ordering information for records in the database. This block is for use by the application; Palm OS® does not use it.

Reading Handheld Database Records and Resources

As shown in Table 4.1, the Sync Manager provides a number of functions for retrieving records and resources from a database on the handheld. Each of these functions reads the record or resource into an object of the `CRawRecordInfo` class.

Before calling each function, you must fill in values for some of the `CRawRecordInfo` object's fields. The function retrieves the record from the handheld and fills in other fields with the record information. The fields required for each function are described in the documentation for each function.

Table 4.1 Functions for reading records from handheld databases

Function	Description
`SyncReadNextModifiedRec()`	An iterator function that retrieves the next modified, archived, or deleted record from an opened record database on the handheld. Each call retrieves the next modified record until all modified records have been returned.
`SyncReadNextModifiedRecInCategory()`	An iterator function that retrieves modified records, including deleted and archived records, in a category from an open database on the handheld. Each call retrieves the next modified record from the category, until all modified records have been retrieved.
`SyncReadNextRecInCategory()`	An iterator function that retrieves any record in a category from an open database on the handheld, including deleted, archived, and modified records. Each call retrieves the next record from the category, until all records have been retrieved.
`SyncReadRecordById()`	Retrieves a record, by ID, from an open database on the handheld.
`SyncReadRecordByIndex()`	Retrieves a record, by index, from an open database on the handheld.
`SyncReadResRecordByIndex()`	Retrieves a resource record, by index, from an open resource database on the handheld.

Listing 4.2 shows a method named `ImportPrefs()`, which calls `SyncReadRecordByIndex()` to read records by index.

Listing 4.2 Reading records from a handheld database

```
BOOL CNetPrefs::ImportPrefs( BYTE hHHDataBase, WORD recIndex )
{
   BYTE  readBuf[ READ_BUF_SIZE ];
   BOOL  importSuccess = TRUE;

   fHDataBase = hHHDataBase;


   // Read from the HH database at index recIndex.
   CRawRecordInfo rrInfo;

   ::memset( &rrInfo, 0, sizeof( rrInfo ) );

   rrInfo.m_FileHandle  = hHHDataBase;
   rrInfo.m_RecIndex    = recIndex;
   rrInfo.m_pBytes      = (BYTE *) &readBuf[ 0 ];
   rrInfo.m_RecSize     = READ_BUF_SIZE;
   rrInfo.m_TotalBytes  = READ_BUF_SIZE;

   long retVal = ::SyncReadRecordByIndex( rrInfo );
   if ( retVal )
   {
      importSuccess = FALSE;
      fPLogMgr->Message( IDS_CANT_READ_DB );
   }
   else
   {
      // The read succeeded, so copy the data onto fCompressedBuffer.
      fHHDBRecID = rrInfo.m_RecId;

      ASSERT( rrInfo.m_RecSize != 0 );

      if ( fCompressedBuffer != NULL ) delete fCompressedBuffer;
      fCompressedBuffer  = new BYTE[ rrInfo.m_RecSize ];
      fCompressedLength  = rrInfo.m_RecSize;

      ::memcpy( (void *) fCompressedBuffer, (void *) rrInfo.m_pBytes,
         fCompressedLength );
   }

   return importSuccess;

} // CNetPrefs::ImportPrefs
```

Iterating Through a Database

The Sync Manager also provides functions that you can use to iterate through a database on the handheld. To iterate through records in a database, you first reset the current index into the database by calling the `SyncResetRecordIndex()` function, and then repeatedly call one of the following retrieval functions:

- `SyncReadNextRecInCategory()` retrieves the next record in a certain record category.
- `SyncReadNextModifiedRec()` retrieves the next record that has been modified (since the last synchronization).
- `SyncReadNextModifiedRecInCategory()` retrieves the next record in a specific record category that has been modified.

The Sync Manager automatically resets the current database iteration index for a database upon opening the database. To reset the index for subsequent iterations, you must call the `SyncResetRecordIndex` function.

Listing 4.3 shows a section of code from that uses `SyncReadNextModifiedRec()` to retrieve each record that has been modified in the handheld database.

Listing 4.3 Using an iterator function to read handheld records

```
CRawRecordInfo rawRecord;

retval = AllocateRawRecordMemory(rawRecord, EXPENSE_RAW_REC_MEM);
   // Read in each modified remote record one at a time.
while (!err && !retval)
{
   if (!(err = SyncReadNextModifiedRec(rawRecord)))
   {
      // Convert from raw record format to CExpenseRecord.
      if (!m_pDTConvert->ConvertFromRemote(remRecord, rawRecord))
      {
         // Synchronize the record obtained from the handheld.
         retval = SynchronizeRecord(remRecord, locRecord,
            backRecord);
      }
      else
         retval = CONDERR_CONVERT_FROM_REMOTE_REC;
   }
   memset(rawRecord.m_pBytes, 0, rawRecord.m_TotalBytes);
   }
if (err && err != SYNCERR_FILE_NOT_FOUND)
   LogBadReadRecord(err);
```

Modifying a Database While Iterating

The Sync Manager does not support the interleaving of iterating through and modifying a database at the same time, which means that you need to structure your logic not to modify a database while iterating through it. Beginning with version 2.0 of Palm OS software, you can safely delete records from a database while iterating through it, *except* when using the SyncPurgeAllRecsInCategory() function.

Writing Handheld Database Records and Resources

The Sync Manager provides several functions for writing records and resources to a database on the handheld. Each of these functions sends record information that you have stored in an object of the `CRawRecordInfo` class. You can use these functions to modify an existing record or to add a new record to a database.

To write a record to a handheld database, fill in certain fields in the raw record object and then call one of the Sync Manager record-writing functions. The function sends the record to a database on the handheld.

You can use the `SyncWriteRec()` function to write a record to a handheld record database, and you can use `SyncWriteResourceRec()` to write a resource to a resource database on the handheld.

Listing 4.4 shows a method named `ExportPrefs()`, which calls the `SyncWriteRec()` function.

Listing 4.4 Writing records to a handheld

```
BOOL CNetPrefs::ExportPrefs( BYTE hHHDataBase )
{
   BOOL  exportSuccess = TRUE;

   fHDataBase = hHHDataBase;

   if ( fCompressedLength != 0 )
   {
      ASSERT( fCompressedBuffer != NULL );

      // write to the HH database.

      CRawRecordInfo  rrInfo;
      ::memset( &rrInfo, 0, sizeof( rrInfo ) );

      rrInfo.m_FileHandle   = hHHDataBase;
      rrInfo.m_RecId        = fHHDBRecID;
      rrInfo.m_pBytes       = fCompressedBuffer;
      rrInfo.m_RecSize      = fCompressedLength;

      long retVal = ::SyncWriteRec( rrInfo );
      if ( retVal )
      {
         exportSuccess = FALSE;
         fPLogMgr->Message( IDS_CANT_WRITE_DB );
      }

   }

   return exportSuccess;

} // CNetPrefs::ExportPrefs
```

Deleting Records in Handheld Databases

When you use a Sync Manager function to delete a record or resource from a database, the record or resource is immediately deleted and removed from the database. This is in contrast to some calls that applications on the handheld make to delete records, which only mark those records for deletion but do not remove them immediately.

NOTE: Sync Manager functions that begin with `Delete` and `Purge` both delete objects *immediately*.

The Sync Manager provides several functions for deleting records or resources from databases. To use these functions, you assign data to specific fields in an object of the `CRawRecordInfo` class. You then call one of the functions described in Table 4.2.

Table 4.2 Functions for deleting single records

Function	Description
`SyncDeleteRec()`	Deletes a record from a record database.
`SyncDeleteResourceRec()`	Deletes a specific resource from a resource database.

You can delete multiple records by calling one of the functions described in Table 4.3.

Table 4.3 Functions for deleting multiple records

Function	Description
`SyncPurgeDeletedRecs()`	Removes all of the records in a record database that have previously been marked as deleted or archived.
`SyncPurgeAllRecs()`	Removes all records in a record database on the handheld.

Table 4.3 Functions for deleting multiple records *(continued)*

Function	Description
SyncPurgeAllRecsInCategory()	Removes all of the records in a specific category from a record database on the handheld.
SyncDeleteAllResourceRec()	Removes all resources from a resource database.

Cleaning Up Records

You can use the function shown in Table 4.4 at the end of your synchronization operations to clear the record status flags.

Table 4.4 Functions to clean up records

Function	Description
SyncResetSyncFlags()	Resets the modified flag of all records in the opened record database on the handheld. Resets the backup date for an opened record or resource database.

Closing Your Database

After you have finished with a handheld database, you must call the SyncCloseDB() or SyncCloseDBEx() function to close it as shown in Listing 4.1. The Sync Manager allows only one database to be open at any time.

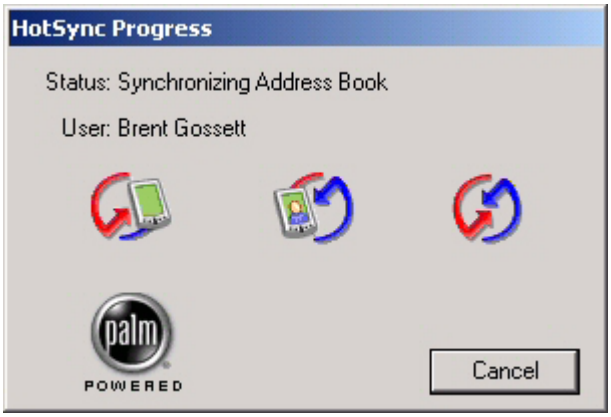
IMPORTANT: If you open a database, you must close it before exiting your conduit; otherwise, other conduits will not be able to open their databases.

Updating the HotSync Manager Progress Display

If your conduit is performing any time-consuming operations between Sync Manager calls, you need to call the `SyncYieldCycles()` function periodically to keep HotSync Manager's progress display indicator current (Figure 4.1). If you neglect to call this function frequently enough, the user interface will appear to be frozen.

You must call the `SyncYieldCycles()` function, and all other Sync Manager functions, from the same thread that launched your conduit.

Figure 4.1 HotSync Manager progress display indicator



5

Using the Extended Sync Manager API

The extended Sync Manager API is a low-level programming interface to HotSync® operations with **extended database**s. This chapter compares this API with that for classic databases described in Chapter 4, "Using the Classic Sync Manager API," on page 29:

The Sync Manager API libraries are located in the `<CDK>\C++\Win\lib` directory of the C/C++ Sync Suite.

Comparing Extended and Classic Sync Manager APIs

The Data Manager in Palm OS Cobalt supports extended record and resource databases. Extended databases differ from or classic databases in several ways. Two of the most important differences are:

- Classic database records, resources, and other data are limited to 64 KB. Extended database data can be much larger, with record sizes up to almost 64 MB.
- Classic, extended, and schema databases exist in disjoint namespaces.
  - Within the *classic* database namespace, database names must be unique.
  - Within the *extended* and *schema* namespaces, database names must be unique *only for a given creator ID.*

  Because the namespaces are disjoint, it is possible for up to three databases, one per namespace, to have the same name and creator ID.

For a comparison of all database types, see "Schema vs. Non-schema Databases" on page 114 in *Introduction to Conduit Development*.

Sync Manager version 2.4 (first shipped with HotSync Manager 6.0) provides an API for accessing extended databases. The *extended Sync Manager API* differs from the classic Sync Manager API in the following ways:

- Extended database API names begin with "SyncDm" instead of "Sync".
- Extended database APIs do not take a **memory card** number parameter. Palm OS Cobalt does not support memory cards.
- Extended database size and offset parameters are 32 bits instead of 16.
- Extended database APIs that need to uniquely identify a database take both a **creator ID** and a **database name**.
- Multiple extended (or schema) databases may be open at the same time; it is also possible to iterate through multiple extended or schema databases at the same time.
- Record modification, deletion, and creation is supported while iterating through an extended database.
- To simplify usage, extended database APIs favor individual parameters over parameter structures like `CRawRecordInfo`. This is a stylistic difference and does not reflect any underlying difference in functionality.
- To provide flexibility for larger data sizes, extended database APIs support partial reads and writes via the addition of an offset parameter; this is consistent with the schema database single column read and write APIs.
- To minimize round trips to the handheld, extended database read APIs automatically cache handheld data on the desktop. This is particularly useful when reading potentially large data of unknown size, because reading such data typically requires dynamic buffer allocation. Dynamic allocation requires two calls: one to retrieve the data size and one to retrieve the actual data. By caching data during the first call, the overhead of a second round trip to the handheld is avoided.

As an example of the difference between the classic and extended Sync Manager APIs, consider the classic SyncWriteRec() API:

```
class CRawRecordInfo {
   public:
   HSByte m_FileHandle;
   UInt32 m_RecId;
   UInt16 m_RecIndex;
   HSByte m_Attribs;
   SInt16 m_CatId;
   SInt32 m_ConduitId;
   UInt32 m_RecSize;
   UInt16 m_TotalBytes;
   HSByte *m_pBytes;
   UInt32 m_dwReserved;
}

SInt32 SyncWriteRec (CRawRecordInfo &rInfo);
```

The corresponding extended database record write API is SyncDmWriteRecord():

```
HSError SyncDmWriteRecord (
   HSByte handle,
   UInt32 *pRecordID,
   UInt16 categoryIndex,
   HSByte attributes,
   UInt32 dataOffset,
   UInt32 dataSize,
   HSBytePtr pRecordData);
```

Note that the supported classic `CRawRecordInfo` members are parameters in the extended API.

Similarly the classic SyncReadRecordById() API:

```
SInt32 SyncReadRecordById (CRawRecordInfo &rInfo);
```

becomes the extended SyncDmReadRecordByID():

```
HSError SyncDmReadRecordByID (
   HSByte handle,
   UInt16 *pRecordIndex,
   UInt32 recordID,
   UInt16 *pCategoryIndex,
```

```
HSByte *pAttributes,
UInt32 dataOffset,
UInt32 *pDataSize,
HSBytePtr pRecordData,
UInt32 *pDataRemaining);
```

A similar transformation yields the extended Sync Manager API for each write and read function.

Data Size

After an extended Sync Manager write operation, the final data size—record size, resource size, etc.—is equal to *dataOffset* + *dataSize*. This implies that all existing data following the specified offset is lost. After a read, the total data size is *dataOffset* + **pDataSize* + **pDataRemaining*. To discover the data size without reading the data, the caller can pass a zero offset and NULL for all output fields except *pDataRemaining*. As noted above, this causes the actual data to be retrieved and cached on the desktop.

6

Using Expansion Technology

This chapter describes how to work with handheld expansion cards and add-on devices from the desktop using the Expansion and Virtual File System (VFS) Managers. The sections in this chapter are:

Expansion Support

Beginning with Palm OS® 4.0, a set of optional system extensions provide a standard mechanism by which Palm OS applications, desktop applications, and conduits can take advantage of the expansion capabilities of various Palm Powered™ handhelds. This capability not only augments the memory and I/O of the handheld, but facilitates data interchange with other Palm Powered handhelds and with devices that are not running Palm OS. These other devices include digital cameras, digital audio players, desktop or laptop computers, and the like.

This section covers the following topics:

Primary vs. Secondary Storage

All Palm Powered handhelds contain **primary storage**—directly addressable memory that is used for both long-term and temporary storage. This includes storage RAM, used to hold nonvolatile user data and applications; and dynamic RAM, which is used as working space for temporary allocations.

On most handhelds, primary storage is contained entirely within the handheld itself. The Palm OS memory architecture does not limit handhelds to this, however; handhelds can be designed to accept additional storage RAM. Some products developed by Handspring work this way; memory modules plugged into the Springboard slot are fully-addressable and appear to a Palm OS application as additional storage RAM.

To access primary storage RAM from the desktop, conduits make Sync Manager API calls during a HotSync® operation (see Chapter 4, "Using the Classic Sync Manager API," on page 29).

Secondary storage, by contrast, is designed primarily to be add-on nonvolatile storage. Although not limited to any particular implementation, most secondary storage media:

- can be inserted and removed from the expansion slot at will
- are based upon a third-party standard, such as Secure Digital (SD), MultiMediaCard (MMC), CompactFlash, Sony's Memory Stick, and others
- present a serial interface, accessing data one bit, byte, or block at a time

Conduits access primary storage through the Sync Manager during a HotSync operation. To access secondary storage, however, conduits use the Expansion and VFS Managers. These have been designed to support as broad a range of serial expansion architectures as possible.

Desktop applications or installers have always used the Install Aide to queue applications and databases for an install conduit to install in primary storage on the handheld during the next HotSync operation. With version 4.0 of the Palm OS platform, they can also use the Install Aide to queue files for an install conduit to copy from the desktop to secondary storage on expansion cards. Similarly, the User Data API enables desktop applications and installers to access information about handheld users on the desktop; now they can also access information about expansion slots on users' handhelds.

For more information about the Install Aide and User Data APIs, see "Using the Install Aide API" on page 128 and "Using the User Data API" on page 145.

Expansion Slot

The expansion slots found on many Palm Powered handhelds vary depending on the manufacturer. While some may accept SD and MMC cards, others may accept Memory Stick or CompactFlash. Note that there is no restriction on the number of expansion slots that handhelds can have.

Depending on the expansion technology used, there can be a wide variety of expansion cards usable with a given handheld:

- Storage cards provide secondary storage and can either be used to hold additional applications and data, or can be used for a specific purpose, for instance as a backup mechanism.
- ROM cards hold dedicated applications and data.
- I/O cards extend the handheld's I/O capabilities. A modem, for instance, could provide wired access, while a Bluetooth transceiver could add wireless capability.
- "Combo" cards provide both additional storage or ROM along with some I/O capability.

Universal Connector

Certain newer Palm Powered handhelds may be equipped with a universal connector that connects the handheld to a HotSync cradle. This connector can be used to connect the handheld to snap-on I/O devices as well. A special slot driver dedicated to this connector allows handheld-to-accessory communication using the serial

portion of the connector. This "plug and play" slot driver presents the peripheral as a card in a slot, even to the extent of providing the card insertion notification when the peripheral is attached.

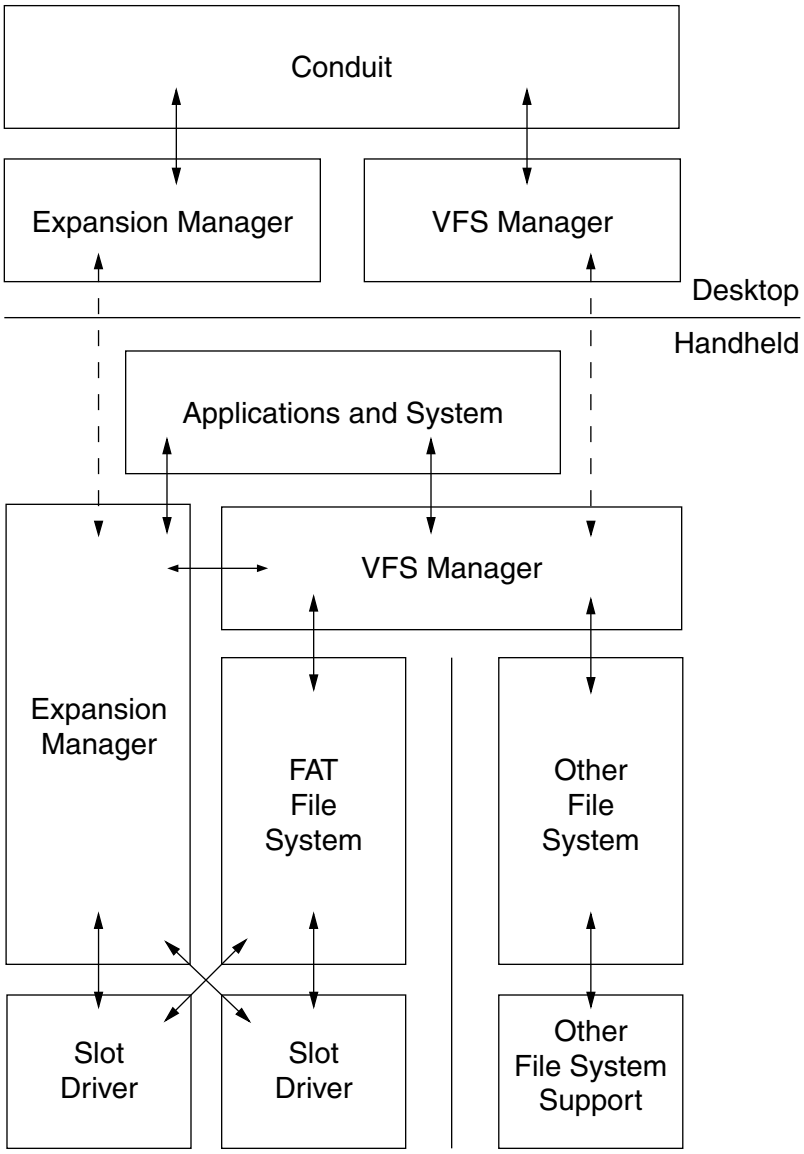
Because the universal connector's slot driver makes a snap-on peripheral appear to be a card in a slot, such peripherals can be treated as expansion cards by the handheld application. From a conduit's perspective, such a peripheral is not accessible during a *cradle-based* HotSync operation, because the universal connector is physically unavailable to a peripheral if it is used by the HotSync cradle. However, if the HotSync operation is via a *wireless* connection—for example, infrared—then a conduit can access such a peripheral. See "Custom Calls" on page 80 for a discussion of how conduits can make custom calls to access custom file systems or I/O devices.

For the remainder of this chapter, wherever an I/O card could be used, the phrase "expansion card" can be taken to mean both "expansion card" and "plug and play peripheral," but only if the the HotSync operation is not via the universal connector.

Architectural Overview

Figure 6.1 illustrates the Palm OS expansion architecture. It is designed to be flexible enough to support multiple file systems and diverse physical expansion mechanisms while still presenting a consistent set of APIs to applications and to other parts of Palm OS. The following sections describe the major components of the Palm OS expansion architecture. Working from the bottom up, those components are:

Figure 6.1 Palm OS expansion architecture



Slot Drivers

A slot driver is a standard Palm OS shared library on the handheld. It is a special library that encapsulates direct access to the hardware and provides a standard set of services to the Expansion Manager and, optionally, to file system libraries. As illustrated in Figure 6.1, neither handheld applications nor conduits normally interact directly with slot drivers.

Each expansion slot has a slot driver associated with it. Slots are identified by a unique **slot reference number**, which is assigned by the Expansion Manager on the handheld. Expansion cards themselves are not numbered individually; applications typically reference the slot into which a card is inserted. Note, however, that a slot may or may not have a card in it at any given time, and that a card can be inserted and removed while an application is running.

The Expansion Manager on the handheld identifies slots by slot reference numbers. These slot reference numbers may change depending on the order in which slot drivers are loaded by the Expansion Manager. Moreover, slot reference numbers are available only to conduits during a HotSync operation via the Expansion Manager API. Therefore HotSync Manager assigns **slot IDs** to slots on the handheld at the beginning of each HotSync operation and saves them for the corresponding user in the user data store on the desktop.

The User Data and Install Aide APIs, which are not used during a HotSync operation, use slot IDs to identify slots instead of slot reference numbers. These APIs simply return the information saved on the desktop during the last HotSync operation, so this information might not be accurate for the next HotSync operation because the user might have changed or updated the handheld between HotSync operations.

File Systems

The Palm OS expansion architecture defines a common interface for all file system implementations on Palm OS. This interface consists of a complete set of APIs for interacting with the file system, including the ability to open, close, read, write, and delete both files and directories on named volumes. Almost all of these handheld

APIs are available on the desktop so that conduits can interact with file systems almost as completely as handheld applications can.

File system implementations are packaged as shared libraries on the handheld. They are modular plug-ins that add support for a particular type of file system, such as VFAT, HFS, or NFS. The Palm OS expansion architecture allows multiple **file system libraries** to be installed at any given time. Typically, an implementation of the VFAT file system is present.

VFAT is the industry standard for flash memory cards of all types. It enables easy transfer of data and or applications to desktops and other devices. The VFAT file system library included with Palm OS software versions 4.0 and later natively supports VFAT file systems on secondary storage media. It is able to recognize and mount FAT and VFAT file systems, and offers to reformat unrecognizable or corrupted media.

Because the VFAT file system requires long filenames to be stored in Unicode/UCS2 format, the standard VFAT file system library supports conversion between UCS2 and Shift-JIS (the standard Palm OS multi-byte character encoding), and between UCS2 and the Palm OS/Latin encoding.

VFS Manager

The Virtual File System (VFS) Manager provides a unified API that gives handheld applications and desktop conduits access to many different file systems on many different media types. It abstracts the underlying file systems so that applications and conduits can be written without regard to the actual file system in use. The VFS Manager provides conduits many API functions for manipulating files, directories, and volumes *during* a HotSync operation.

The VFS Manager and the Sync Manager APIs

With the addition of the VFS Manager to the C/C++ Sync Suite, there are now two distinct ways that conduits can store and retrieve Palm OS user data:

- The Sync Manager accesses resource and record databases in the handheld's primary storage RAM. It effectively calls the handheld's Data Manager, which was specifically designed to make the most of the limited dynamic RAM and the

nonvolatile RAM used instead of disk storage on most handhelds. Use the Sync Manager to store and retrieve Palm OS user data when storage on the handheld is all that is needed, or when efficient access to data is paramount.

- The VFS and Expansion Managers were designed specifically to support many types of expansion memory as secondary storage. The VFS Manager API presents a consistent interface to many different types of file systems on many types of external media. Conduits that use the VFS Manager API can support the widest variety of file systems. Use the VFS Manager when your conduit needs to read and write data stored on external media.

Conduits should use the appropriate APIs for each given situation. The Sync Manager, being an efficient manager of storage in the storage heap, should be used whenever access to external media is not absolutely needed. Use the VFS Manager API when interoperability and file system access is needed. Note, however, that the VFS Manager adds the extra overhead of buffering all reads and writes in the handheld's memory when accessing data, so only conduits that specifically need this functionality should use the VFS Manager.

For more information on the Sync Manager see Chapter 4, "Using the Classic Sync Manager API," on page 29. For details of the API presented by the VFS Manager, see Chapter 10, "Virtual File System Manager API," on page 561 in the *C/C++ Sync Suite Reference*.

Expansion Manager

The Expansion Manager is a software layer that manages slot drivers on Palm Powered handhelds. Supported expansion card types include, but are not limited to, Memory Stick and SD cards. The Expansion Manager does not support these expansion cards directly; rather, it provides an architecture and higher level set of API functions that, with the help of low-level slot drivers and file system libraries, support these types of media.

The Expansion Manager on the handheld:

- broadcasts notification of card insertion and removal
- plays sounds to signify card insertion and removal
- mounts and unmounts card-resident volumes

The Expansion Manager API for conduits provides an interface to the Expansion Manager on the handheld *during* a HotSync operation. Through this interface, conduits can determine whether an expansion card is present in a slot and get information about those cards.

For details of the API presented by the Expansion Manager, see Chapter 9, "Expansion Manager API," on page 547 in the *C/C++ Sync Suite Reference*.

Standard Directories

The user experience presented by Palm OS is simpler and more intuitive than that of a typical desktop computer. Part of this simplicity arises from the fact that Palm OS does not present a file system to the user. Users do not have to understand the complexities of a typical file system; applications are readily available with one or two taps of a button or icon, and data associated with those applications is accessible only through each application. Maintaining this simplicity of user operation while supporting a file system on an expansion card is made possible through a standard set of directories on the expansion card.

Table 6.1 lists the standard directory layout for all "standards compliant" Palm OS secondary storage. All Palm OS relevant data

should be in the `/PALM` directory (or in a subdirectory of the `/PALM` directory), effectively partitioning off a private name space.

Table 6.1 Standard directories

Directory	Description
`/`	Root of the secondary storage.
`/PALM`	Most data written by Palm OS applications lives in a subdirectory of this directory. `start.prc` lives directly in `/PALM`. This optional file is automatically run when the secondary storage volume is mounted. Other applications may also reside in this directory.
`/PALM/Backup`	Reserved by Palm OS for backup purposes.
`/PALM/Programs`	Catch-all for other applications and data.
`/PALM/Launcher`	Home of Launcher-visible applications.

In addition to these standard directories, the VFS Manager supports the concept of a **default directory**; a directory in which data of a particular type is typically stored. See "Determining the Default Directory for a Particular File Type" on page 77 for more information.

Card Insertion and Removal

The Expansion Manager supports the insertion and removal of expansion media at any time. The handheld continues to run as before, though an application switch may occur upon card insertion. The handheld need not be reset or otherwise explicitly informed that a card has been inserted or removed.

WARNING! Because of the way certain expansion cards are constructed, if the user removes an expansion card while an application or conduit is *writing* to it, in certain rare circumstances it is possible for the card to become damaged to the point where either it can no longer be used or it must be reformatted. To the greatest extent possible, applications and conduits should write to the card only at well-defined points. The card can be removed without fear of damage while an application or conduit is *reading* from it, however.

If the user removes a card while your conduit is accessing it during a HotSync operation, the Expansion Manager or VFS Manager function fails. PalmSource, Inc. recommends that your conduit always check errors returned by all API functions and add helpful messages to the user in the HotSync log.

Checking for Expansion Cards

Before looking for an expansion card, your conduit should first make sure that the handheld supports expansion by verifying the presence of the Expansion and VFS Managers. It can then query for mounted volumes. Finally, your conduit may want to ascertain the capabilities of the card; whether it has memory, whether it does I/O, and so on. The following sections describe each of these steps:

Verifying Handheld Compatibility

There are many different types of Palm Powered handhelds, and in the future there will be many more. Some will have expansion slots to support secondary storage, and some will not. Hardware to support secondary storage is optional, and may or may not be present on a given type of handheld. Because the Expansion and VFS Managers are of no use on a handheld that has no physical expansion capability, they are optional system extensions that are not present on every type of Palm Powered handheld.

Because of the great variability both in handheld configuration and in the modules that can be plugged into or snapped onto the handheld, conduits should not attempt to detect the manufacturer or model of a specific handheld when determining whether it supports secondary storage. Instead, check for the presence and capabilities of the underlying operating system.

The VFS Manager and the Expansion Manager on the handheld are individual system extensions that are both optional. They both make use of other parts of the operating system that were introduced in Palm OS software version 4.0.

NOTE: Although your conduit could check for the presence of both the VFS and Expansion Managers, it can take advantage of the fact that the VFS Manager relies on the Expansion Manager and is not present without it. Therefore, if the VFS Manager is present, you can safely assume that the Expansion Manager is present as well.

Therefore your conduit must check for the presence of the VFS Manager on the handheld before calling any VFS or Expansion Manager API functions. This section presents two methods of verifying the presence of the VFS Manager on the handheld:

- Using the VFSSupport Function
- Using the SyncReadFeature Function

Using the VFSSupport Function

The desktop VFS Manager API provides the `VFSSupport()` function to verify whether the handheld has an expansion slot and whether any file systems are present.

When you call `VFSSupport()`, it passes back the version of the Expansion Manager on the handheld—which is identical to the VFS Manager version number—or zero, if no expansion slot is present. In a second parameter, it passes back the number of volumes present on the card or zero, if there is no file system present on the card or no card in the slot.

NOTE: The `VFSSupport()` function is the primary method for conduits to determine whether a handheld has expansion slots. This information has already been obtained by the desktop VFS Manager, so no additional calls are made to the handheld at the time your conduit calls `VFSSupport()`—which makes it more efficient than using `SyncReadFeature()`.

Using the SyncReadFeature Function

The Sync Manager API provides `SyncReadFeature()` to verify the presence of defined feature sets on the handheld. To check for the VFS Manager's system feature, do the following:

- call `SyncReadFeature()`
- supply `sysFileCVFSMgr` for the feature creator
- supply `vfsFtrIDVersion` for the feature number

Listing 6.1 shows how to use `SyncReadFeature()` to check for the presence and proper version of the VFS Manger. Note that `expectedVFSMgrVersionNum` should be replaced by the actual version number you expect.

Listing 6.1 SyncReadFeature to verify the presence of the VFS Manager

```
UINT32 vfsMgrVersion;
long err;

err = SyncReadFeature(sysFileCVFSMgr, vfsFtrIDVersion,
                      &vfsMgrVersion);
if(err){
    // The VFS Manager is not installed.
       }
  else {
    // Check the version number of the VFS Manager,
    // if necessary.
    if(vfsMgrVersion == expectedVFSMgrVersionNum)
        // Everything is OK.
}
```

Even though presence of the VFS Manager implies presence of the Expansion Manager, you can also check for the Expansion Manager's system feature as follows:

- call `SyncReadFeature()`
- supply `sysFileCExpansionMgr` for the feature creator
- supply `expFtrIDVersion` for the feature number

Listing 6.2 shows how to use `SyncReadFeature()` to check for the presence and proper version of the Expansion Manger. Note that `expectedExpMgrVersionNum` should be replaced by the actual version number you expect.

Listing 6.2 SyncReadFeature to verify the presence of the Expansion Manager

```
UINT32 expMgrVersion;
long err;

err = SyncReadFeature(sysFileCExpansionMgr, expFtrIDVersion,
                      &expMgrVersion);
if(err){
    // The Expansion Manager is not installed.
       }
  else {
    // Check version number of the Expansion Manager,
    // if necessary.
    if(expMgrVersion == expectedExpMgrVersionNum)
        // Everything is OK.
}
```

Checking for Mounted Volumes

Conduits rely on the fact that Palm OS automatically mounts any recognized volumes inserted into or snapped onto the handheld. Therefore conduits can simply enumerate the mounted volumes and select one as appropriate. Listing 6.3 illustrates how to do this.

Listing 6.3 Enumerating mounted volumes

```
WORD numVolumes = 0;
WORD *pwVolRefNumList;

// The first call returns only the number of mounted volumes,
// not their reference numbers.
VFSVolumeEnumerate (&numVolumes, NULL);
  if (numVolumes)
  {
    // Allocate buffer for volume reference list.
    pwVolRefList = new WORD [numVolumes];
  if (pwVolRefList != NULL)
    {
      // Get the volume reference numbers.
    VFSVolumeEnumerate (&numVolumes, pwVolRefList);
    }
}
```

The volume reference numbers obtained from `VFSVolumeEnumerate()` can then be used with many of the volume, directory, and file operations that are described later in this chapter.

Occasionally a conduit needs to know more than that there is secondary storage available for use. Those conduits likely need to take a few extra steps, beginning with checking each of the handheld's slots, as described in the next section.

Enumerating Slots

Before you can determine which expansion modules are attached to a handheld, you must first determine how those modules could be attached. Expansion cards and some I/O devices could be plugged into physical slots, and snap-on modules could be connected through the handheld's universal connector. Irrespective of how they are physically connected, the Expansion Manager presents these to the developer as slots. The `ExpSlotEnumerate()` function makes it simple to enumerate these slots. Listing 6.4 illustrates the use of this function.

Listing 6.4 Enumerate a handheld's expansion slots

```
WORD wSlotRefList[32]; // Buffer for slot reference numbers.
WORD wSlotRefCount;    // Number of entries allocated for list.
long retval;

// Allocate enough space for buffer.
wSlotRefCount = sizeof (wSlotRefList) / sizeof (wSlotRefList[0]);
retval = ExpSlotEnumerate(&wSlotRefCount, wSlotRefList, NULL);
```

The array of slot reference numbers passed back by `ExpSlotEnumerate()` uniquely identify all slots. A slot reference number can be supplied to various Expansion Manager functions to obtain information about the slot, such as whether there is a card or other expansion module present in the slot.

Checking a Slot for the Presence of a Card

Use the `ExpCardPresent()` function to determine whether a card is present in a given slot. Given the slot reference number, this function returns `SYNCERR_NONE` if there is a card in the slot, or an error if either there is no card in the slot or there is a problem with the specified slot.

Determining a Card's Capabilities

Just knowing that an expansion card is inserted into a slot or connected to the handheld is not enough; your conduit needs to know something about the card to ensure that the operations it needs to perform are compatible with the card. For instance, if your conduit needs to write data to the card, its important to know whether writing is permitted.

The capabilities available to your conduit depend not only on the card but on the slot driver as well. Handheld manufacturers will provide one or more slot drivers that define standard interfaces to certain classes of expansion hardware. Card and device manufacturers may also choose to provide card-specific slot drivers, or they may require that applications use the slot custom control function and a registered creator code to access and control certain cards.

The slot driver is responsible for querying expansion cards for a standard set of capabilities. When a slot driver is present for a given expansion card, you can use the `ExpCardInfo()` function to determine the following:

- the name of the expansion card's manufacturer
- the name of the expansion card
- the "device class," or type of expansion card. Values returned here might include "Ethernet" or "Backup"
- a unique identifier for the device, such as a serial number
- whether the card supports both reading and writing, or whether it is read-only

Note that the existence of the `ExpCardInfo()` function does not imply that all expansion cards support these capabilities. It means only that the slot driver is able to assess a card and report its findings up to the Expansion Manager.

Volume Operations

If an expansion card supports a file system, the VFS Manager allows you to perform a number of standard volume operations. To determine which volumes are currently mounted and available, use `VFSVolumeEnumerate()`. This function, the use of which is illustrated in "Checking for Mounted Volumes" on page 63, returns a list of volume reference numbers, which you select from and then supply to the remainder of the volume operations.

NOTE: Volume reference numbers can change each time the handheld mounts a given volume. To keep track of a particular volume, save the volume's label rather than its reference number.

When the user inserts a card containing a mountable volume into a slot, the VFS Manager attempts to mount the volume automatically. Conduits cannot otherwise mount a volume.

Use `VFSVolumeFormat()` to format a volume. Once the card has been formatted, the VFS Manager automatically mounts it and `VFSVolumeFormat()` passes back a new volume reference number.

The `VFSVolumeGetLabel()` and `VFSVolumeSetLabel()` functions get and set the volume label, respectively. Because the file system is responsible for verifying the validity of strings, you can try to set the volume label to any desired value. If the file system does not natively support the name given, the VFS Manager creates the `/VOLUME.NAM` file used to support long volume names (see "Naming Volumes" on page 69 for more information) or you get an error back if the file system does not support the supplied string.

NOTE: Most conduits should not need to call the `VFSVolumeSetLabel()` function. This function may create or delete a file in the root directory, which would invalidate any current calls to `VFSDirEntryEnumerate()`.

Additional information about the volume can be obtained through the use of `VFSVolumeSize()` and `VFSVolumeInfo()`. As the name implies, `VFSVolumeSize()` returns size information about the volume. In particular, it returns both the total amount of space on the volume, in bytes, and the amount of that volume's space that is currently in use, again in bytes. `VFSVolumeInfo()` returns various pieces of information about the volume, including:

- whether the volume is hidden
- whether the volume is read-only
- whether the volume is supported by a slot driver, or is being simulated by Palm OS Emulator
- the type and creator of the underlying file system
- the slot with which the volume is associated, and the reference number of the slot driver controlling the slot
- the type of media on which this volume is located, such as SD, CompactFlash, or Memory Stick

All of the above information is returned encapsulated within a `VolumeInfoType` structure. Whether the volume is hidden or read-only is further encoded into a single field within this structure; see "Volume Attributes" on page 578 in the *C/C++ Sync Suite Reference* for the bits that make up this field.

The rest of this section covers the following topics:

Hidden Volumes

Included among the volume attributes is a "hidden" bit, `vfsVolumeAttrHidden`, that indicates whether the volume on the card is to be visible or hidden. Hidden volumes are typically not meant to be directly available to the user; the Launcher and the CardInfo application both ignore all hidden volumes.

To make a volume hidden, simply create an empty file named `HIDDEN.VOL` in the `/PALM` directory. The `VFSVolumeInfo()` function looks for this file and, if found, returns the `vfsVolumeAttrHidden` bit along with the volume's other attributes.

Matching Volumes to Slots

In most cases, a conduit does not need to know the specifics of an expansion card as provided by the `ExpCardInfo()` function. Often, the information provided by the `VFSVolumeInfo()` function is enough. Some applications need to know more about a particular volume, however. The name of the manufacturer or the type of card, for instance, may be important.

The `VolumeInfoType` structure returned from `VFSVolumeInfo()` contains a `slotRefNum` field that can be passed to `ExpCardInfo()`. This allows you to obtain specific information about the card on which a particular volume is located.

Obtaining volume information that corresponds to a given slot reference number is not quite so simple, because there is no function that returns the volume reference number given a slot reference number. If the volumes are slot-based, you can, however, iterate through the mounted volumes and check each volume's slot reference number. (To determine whether a volume is slot-based, in the `VolumeInfoType` structure check whether `mountClass` is set

to `sysFileTSlotDriver` before accessing the volume specified by `slotRefNum`.)

Naming Volumes

Different handheld file system libraries support volume names of different maximum lengths and have different restrictions on character sets. The file system library is responsible for verifying whether or not a given volume name is valid, and returns an error if it is not. From a conduit developer's standpoint, volume names can be up to 255 characters long, and can include any printable character.

The file system library is responsible for translating the volume name into a format that is acceptable to the underlying file system. For example, when the 8.3 naming convention is used to translate a long volume name, the first eleven valid, non-space characters are used. Valid characters in this instance are A-Z, 0-9, $, %, ', -, _, @, ~, `, !, (,), ^, #, and &.

For more information on naming volumes on handheld expansion cards, see the *Palm OS Programmer's Companion*.

File Operations

All of the familiar operations you use to operate on files in a desktop application are supported by the VFS Manager for handheld expansion cards; these are listed in "Common File Operations." In addition, the VFS Manager includes a set of functions that simplify the way you work with files that represent Palm OS record databases (PDB) or resource databases (PRC). These are covered in "Working with Palm OS Databases" on page 73.

This section covers the following topics:

Common File Operations

As shown in Table 6.2, the VFS Manager provides all of the standard file operations that should be familiar from desktop and larger computer systems. Because these functions work largely as you would expect, their use is not detailed here. See the descriptions of each individual function in Chapter 10, "Virtual File System Manager API," on page 561 in the *C/C++ Sync Suite Reference* for the parameters, return values, and side effects of each.

Note that some of these functions can be applied to both files and directories, while others work only with files.

Table 6.2 Common file operations

Function	Description
`VFSFileOpen()`	Opens a file, given a volume reference number and a file path.
`VFSFileClose()`	Closes an open file.
`VFSFileRead()`	Reads data from a file into the specified buffer.
`VFSFileWrite()`	Writes data to an open file.
`VFSFileSeek()`	Sets the position within an open file from which to read or write.

Table 6.2 Common file operations *(continued)*

Function	Description
VFSFileTell()	Gets the current position of the file pointer within an open file.
VFSFileEOF()	Gets the end-of-file status for an open file.
VFSFileCreate()	Creates a file, given a volume reference number and a file path.
VFSFileDelete()	Deletes a closed file.
VFSFileRename()	Renames a closed file.
VFSFileSize()	Gets the size of an open file.
VFSFileResize()	Changes the size of an open file.
VFSFileGetAttributes()	Gets the attributes of an open file, including hidden, read-only, system, and archive bits. See "File and Directory Attributes" on page 572 in the *C/C++ Sync Suite Reference* for the bits that make up the attributes field.
VFSFileSetAttributes()	Sets the attributes of an open file, including hidden, read-only, system, and archive bits.
VFSFileGetDate()	Gets the created, modified, and last accessed dates for an open file.
VFSFileSetDate()	Sets the created, modified, and last accessed dates for an open file.

Once a file has been opened, it is identified by a unique **file reference number**: a FileRef. Functions that work with open files take a file reference. Others, such as VFSFileOpen(), require a volume reference and a path that identifies the file within the volume. Note that all paths are volume relative, *and absolute within that volume:* the VFS Manager has no concept of a "current working directory," so relative path names are not supported. The directory separator character is the forward slash: "/". The root directory for the specified volume is specified by a path of "/".

Naming Files

Different file systems support filenames and paths of different maximum lengths. The file system library is responsible for verifying whether or not a given path is valid and returns an error if it is not valid. From a conduit developer's standpoint, filenames can be up to 255 characters long and can include any normal character including spaces and lowercase characters in any character set. They can also include the following special characters:

```
$ % ' - _ @ ~ '! ( ) ^ # & + , ; = [ ]
```

The file system library is responsible for translating each filename and path into a format that is acceptable to the underlying file system. For example, when the 8.3 naming convention is used to translate a long filename, the following guidelines are used:

- The name is created from the first six valid, non-space characters which appear before the last period. The only valid characters are A-Z, 0-9, $, %, ', -, _, @, ~, ', !, (,), ^, #, and &.
- The extension is the first three valid characters after the last period.
- The end of the six byte name has "~1" appended to it for the first occurrence of the shortened filename. Each subsequent occurrence uses the next unique number, so the second occurrence would have "~2" appended, and so on.

The standard VFAT file system library provided with all Palm Powered handhelds that support expansion uses the above rules to create FAT-compliant names from long filenames.

Working with Palm OS Databases

Expansion cards are often used to hold Palm OS applications and data in PRC and PDB format. Because of the way that secondary storage media are connected to the Palm Powered handheld, applications cannot be run directly from the expansion card, nor can conduits manipulate databases using the Sync Manager without first transferring them to main memory with the VFS Manager. Conduits written to use the VFS Manager, however, can operate directly on files located on an expansion card.

NOTE: Whenever possible give the same name to the PRC/PDB file and to the database. If the PRC/PDB filename differs from the database name, and the user copies your database from the card to the handheld and then to another card, the filename may change. This is because the database name is used when a database is copied from the handheld to the card.

Transferring Palm OS Databases to and from Expansion Cards

The `VFSExportDatabaseToFile()` function converts a database from its internal format on the handheld to its equivalent PRC or PDB file format and transfers it to an expansion card. The `VFSImportDatabaseFromFile()` function does the reverse; it transfers the `.prc` or `.pdb` file to primary storage memory and converts it to the internal format used by Palm OS. Use these functions when moving Palm OS databases between main memory and an expansion card.

The `VFSExportDatabaseToFile()` and `VFSImportDatabaseFromFile()` routines are atomic and, depending on the size of the database and the mechanism by which it is being transferred, can take some time.

IMPORTANT: These import and export functions do not work with schema databases.

Directory Operations

All of the familiar operations you would use to operate on directories are supported by the VFS Manager; these are listed in "Common Directory Operations" on page 75. One common operation—determining the files that are contained within a given directory—is covered in some detail in "Enumerating the Files in a Directory" on page 76. To improve data interchange with devices that are not running Palm OS, expansion card manufacturers have specified default directories for certain file types. "Determining the Default Directory for a Particular File Type" on page 77 discusses how you can both determine and set the default directory for a given file type.

This section covers the following topics:

Directory Paths

All paths are volume relative, *and absolute within that volume*: the VFS Manager has no concept of a "current working directory," so relative path names are not supported. The directory separator character is the forward slash: "/". The root directory for the specified volume is specified by a path of "/".

See "Naming Files" on page 72 for details on valid characters to use in file and directory names.

Common Directory Operations

As shown in Table 6.3, the VFS Manager provides all of the standard directory operations that should be familiar from desktop and larger computer systems. Because these functions work largely as you would expect, their use is not detailed here. See the descriptions of each individual function Chapter 10, "Virtual File System Manager API," on page 561 in the *C/C++ Sync Suite Reference* for the parameters, return values, and side effects of each.

Note that most of these functions can be applied to files as well as directories.

Table 6.3 Common directory operations

Function	Description
`VFSDirCreate()`	Creates a new directory.
`VFSFileDelete()`	Deletes a directory, given a path.
`VFSFileRename()`	Renames a directory.
`VFSFileOpen()`	Opens the specified directory.
`VFSFileClose()`	Closes the specified directory.
`VFSFileGetAttributes()`	Gets the attributes of an open directory, including hidden, read-only, system, and archive bits. See "File and Directory Attributes" on page 572 in the *C/C++ Sync Suite Reference* for the bits that make up the attributes field.
`VFSFileSetAttributes()`	Set the attributes of an open directory, including hidden, read-only, system, and archive bits.
`VFSFileGetDate()`	Get the created, modified, and last accessed dates for an open file.
`VFSFileSetDate()`	Set the created, modified, and last accessed dates for an open file.

Enumerating the Files in a Directory

Enumerating the files within a directory is made with the VFSDirEntryEnumerate() function. Listing 6.5 illustrates the use of this function. Note that volRefNum must be declared and initialized prior to the following code.

Listing 6.5 Enumerating a directory's contents

```
// Open the directory and iterate through the files in it.
// volRefNum must have already been defined.
FileRef dirRef;

err = VFSFileOpen (volRefNum, "/", vfsModeRead, &dirRef);
if(err == errNone) {
  // Iterate through all the files in the open directory
  UInt32 fileIterator;
  FileInfoType fileInfo;
  char *fileName = new char[256];   // Should check for err.

  fileInfo.nameP = fileName;        // Point to local buffer.
  fileInfo.nameBufLen = sizeof(fileName);
  fileIterator = expIteratorStart;
  while (fileIterator != expIteratorStop) {
    // Get the next file
    err = VFSDirEntryEnumerate(dirRef, &fileIterator,
                               &fileInfo);
    if(err == errNone) {
      // Process the file here.
    }
  } else {
    // Handle directory open error here.
  }
  delete [] fileName;
}
```

Each time through the while loop, VFSDirEntryEnumerate() sets the FileInfoType structure as appropriate for the file currently being enumerated. Note that if you want the filename, it is not enough to simply allocate space for the FileInfoType structure; you must also allocate a buffer for the filename, set the appropriate pointer to it in the FileInfoType structure, and specify your buffer's length. Because the only other information

encapsulated within `FileInfoType` is the file's attributes, most conduits also want to know the file's name.

Determining the Default Directory for a Particular File Type

As explained in "Standard Directories" on page 57, the expansion capabilities of Palm OS include a mechanism to map MIME types or file extensions to specific directory names. This mechanism is specific to the slot driver: where an image might be stored in the "`/Images`" directory on a Memory Stick, on an MMC card it may be stored in the "`/DCIM`" directory. The VFS Manager includes a function that enables your conduit to get the default directory on a particular volume for a given file extension or MIME type; unlike handheld applications, conduits cannot register and unregister their own default directories.

The `VFSGetDefaultDirectory()` function takes a volume reference and a string containing the file extension or MIME type and returns a string containing the full path to the corresponding default directory. When specifying the file type, either supply a MIME media type/subtype pair, such as "image/jpeg", "text/plain", or "audio/basic"; or a file extension, such as "`.jpeg`". As with most other such VFS Manager functions, you must pre-allocate a buffer to contain the returned path. Supply a pointer to this buffer along with the buffer's length. The length is updated upon return to indicate the actual length of the path, which will not exceed the originally-specified buffer length.

The default directory registered for a given file type is intended to be the "root" default directory. If a given default directory has one or more subdirectories, conduits should also search those subdirectories for files of the appropriate type.

`VFSGetDefaultDirectory()` allows you to determine the directory associated with a particular file suffix. However, there is no way to get the entire list of file suffixes that are mapped to default directories.

Default Directories Registered at Initialization

The VFS Manager registers the file types in Table 6.4 under the `expMediaType_Any` media type, which `VFSGetDefaultDirectory()` reverts to when there is no default registered by the slot driver for a given media type.

Table 6.4 Default directory registrations

File Type	Path
`.prc`	`/PALM/Launcher/`
`.pdb`	`/PALM/Launcher/`
`.pqa`	`/PALM/Launcher/`
`application/vnd.palm`	`/PALM/Launcher/`
`.jpg`	`/DCIM/`
`.jpeg`	`/DCIM/`
`image/jpeg`	`/DCIM/`
`.gif`	`/DCIM/`
`image/gif`	`/DCIM/`
`.qt`	`/DCIM/`
`.mov`	`/DCIM/`
`video/quicktime`	`/DCIM/`
`.avi`	`/DCIM/`
`video/x-msvideo`	`/DCIM/`
`.mpg`	`/DCIM/`
`.mpeg`	`/DCIM/`
`video/mpeg`	`/DCIM/`
`.mp3`	`/AUDIO/`
`.wav`	`/AUDIO/`
`audio/x-wav`	`/AUDIO/`

The SD slot driver provided by PalmSource, Inc. registers the file types in Table 6.5, because it has an appropriate specification for these file types.

Table 6.5 Directories registered by the SD slot driver

File Type	Path
`.jpg`	`/DCIM/`
`.jpeg`	`/DCIM/`
`image/jpeg`	`/DCIM/`
`.qt`	`/DCIM/`
`.mov`	`/DCIM/`
`video/quicktime`	`/DCIM/`
`.avi`	`/DCIM/`
`video/x-msvideo`	`/DCIM/`

Although the directories registered by this SD slot driver all happen to be duplicates of the default registrations made by the VFS Manager, they are also registered under the SD media type because the SD specification explicitly includes them.

Slot drivers written by other Palm Powered handheld manufacturers that support different media types, such as Memory Stick, register default directories appropriate to their media's specifications. In some cases these registrations may override the `expMediaType_Any` media type registration, or in some cases augment the `expMediaType_Any` registrations with file types not previously registered.

These registrations are intended to aid applications developers, but you are not required to follow them. Although you can choose to ignore these registrations, by following them you improve interoperability between applications and other devices. For example, a digital camera which conforms to the media specifications puts its pictures into the registered directory (or a subdirectory of it) appropriate for the image format and media type. By looking up the registered directory for that format, an image

viewer application on the handheld can easily find the images without having to search the entire card. These registrations also help prevent different developers from hard-coding different paths for specific file types. Thus, if a user has two different image viewer applications, both look in the same location and find the same set of images.

Registering these file types at initialization allows the default card install conduit (`CardInst.dll`) to transfer files of these types to an expansion card. During the HotSync process, files of the registered types are placed directly in the specified directories on the card.

Custom Calls

Recognizing that some file systems may implement functionality not covered by the APIs included in the VFS and Expansion Managers, the VFS Manager includes a single function that exists solely to give developers access to the underlying file system. This function, `VFSCustomControl()`, takes a registered creator ID and a selector that together identify the operation that is to be performed. `VFSCustomControl()` can either request that a specific file system perform the specified operation, or it can iterate through all of the currently-registered file systems in an effort to locate one that responds to the desired operation.

Parameters are passed to the file system's custom function through a single `VFSCustomControl()` parameter. This parameter, `pDataBuf`, is declared as a `void *` so you can pass a pointer to a structure of any type. A second parameter, `valueLenP`, allows you to specify the length of `pDataBuf`. Note that these values are simply passed to the file system and are in reality dependent upon the underlying file system. See the description of `VFSCustomControl()` in the *C/C++ Sync Suite Reference* for more information.

Because `VFSCustomControl()` is designed to allow access to nonstandard functionality provided by a particular file system, see the documentation provided with that file system for a list of any custom functions that it provides.

The rest of this section discusses "Custom I/O."

Custom I/O

While the Expansion and VFS Managers provide higher-level OS support for secondary storage, they do not attempt to present anything more than a raw interface to custom I/O applications. Because it is not possible to envision all uses of an expansion mechanism, the Expansion and VFS Managers simply try to get out of the way of custom hardware.

The Expansion Manager provides insertion and removal notification and can load and unload drivers. Everything else is the responsibility of the application developer. PalmSource, Inc. has defined a common expansion slot driver API which is extensible by Palm OS licensees. This API is designed to support all of the needs of the Expansion Manager, the VFS Manager, and the file system libraries. Conduits that need to communicate with an I/O device, however, may need to go beyond the provided APIs. Such conduits should wherever possible use the slot custom call, which provides direct access to the expansion slot driver. See the developer documentation provided to licensees for more information on slot drivers and the slot custom call. For documentation on functions made available by a particular I/O device, along with how you access those functions, contact the I/O device manufacturer.

NOTE: If a custom I/O device connects to the handheld via the same connector as the HotSync cradle, the custom I/O device is, of course, not accessible during a *cradle-based* HotSync operation. However, if the HotSync operation is via a *wireless* connection—for example, infrared—then a conduit can access such a device.

Debugging

Palm OS Simulator supports the expansion capabilities of the VFS Manager. It presents a directory on the host file system as a volume to the virtual file system. You can populate this directory on your host system and then simulate a volume mount. Changes made to the simulated expansion card's contents can be verified simply by examining the directory on the host.

For more information on configuring and operating Palm OS Simulator, see *Palm OS Cobalt Simulator Guide* or *Testing with Palm OS Garnet Simulator*. For information on performing HotSync operations with Simulator, see Chapter 8, "Synchronizing with Palm OS Simulator," on page 49 of the *Conduit Development Utilities Guide*.

Summary of Expansion and VFS Managers

Expansion Manager Functions

ExpCardInfo()
ExpCardPresent()
ExpSlotEnumerate()
ExpSlotMediaType()

VFS Manager Functions

Working with Files

VFSFileClose()
VFSFileCreate()
VFSFileDelete()
VFSFileEOF()
VFSFileGet()
VFSFileGetAttributes()
VFSFileGetDate()
VFSFileOpen()
VFSFilePut()
VFSFileRead()
VFSFileRename()
VFSFileResize()
VFSFileSeek()
VFSFileSetAttributes()
VFSFileSetDate()
VFSFileSize()
VFSFileTell()
VFSFileWrite()

Working with Directories

VFSDirCreate()
VFSDirEntryEnumerate()
VFSFileClose()
VFSFileDelete()
VFSFileGetAttributes()
VFSFileGetDate()
VFSFileOpen()
VFSFileRename()
VFSFileSetAttributes()
VFSFileSetDate()
VFSGetDefaultDirectory()

Working with Volumes

VFSVolumeEnumerate()
VFSVolumeFormat()
VFSVolumeGetLabel()
VFSVolumeInfo()
VFSVolumeSetLabel()
VFSVolumeSize()
VFSSupport()

Miscellaneous Functions

VFSCustomControl()
VFSExportDatabaseToFile()
VFSImportDatabaseFromFile()
VFSGetAPIVersion()
VFSSupport()

7

Debugging Conduits

This chapter provides information to help you debug your conduit, including the following:

Adding Additional Logging Support to Your Conduit

You can launch HotSync Manager with several different flags that increase the level of detail in the HotSync log file. These flags are listed in Chapter 4, "Using Command-line Options for HotSync Manager," on page 24 in the *Conduit Development Utilities Guide*.

You can easily add support in your conduit for the HotSync debugging flags by calling the Windows API function `GetCommandLine()` and parsing the argument list, as shown in Listing 7.1. PalmSource, Inc. recommends that you at least support the verbose (`-v`) option, as this can help you to track problems and troubleshoot technical support calls.

Listing 7.1 Extracting the HotSync.exe command line arguments

```
#define LOG_NORMAL  0x0000
#define LOG_L1      0x0001
#define LOG_L2      0x0002
#define LOG_VERBOSE 0x0004

DWORD dwLogLevel;

struct logOpt_t
{
   char *szCmd;
   DWORD dwCmdValue;
} logOptions[] = {
   _T("V"), LOG_VERBOSE,
   _T("L1"), LOG_L1,
   _T("L2"), LOG_L2
};

void ProcessCmdLineParameters()
{
   CString csCmdLine(GetCommandLine());
   csCmdLine.MakeUpper();

   CString csDelim = _T("-");

   dwLogLevel = LOG_NORMAL;
  int nSize = sizeof(logOptions)/sizeof(logOpt_t);

   for (int n=0; n<nSize; n++)
   {
      if (csCmdLine.Find(csDelim + logOptions[n].szCmd) != -1)
         dwLogLevel |= logOptions[n].dwCmdValue;
   }
}
```

Common Troubleshooting Help

This section describes solutions to several common problems that developers have with getting their conduits working. See Appendix A, "Debugging Tips," on page 59 in the *Conduit Development Utilities Guide* for other useful information.

TIP: You can use the Conduit Inspector utility that ships with the CDK to get more information about your conduit when HotSync Manager tries to load it. See Chapter 6, "Conduit Inspector Utility," on page 29 in the *Conduit Development Utilities Guide*.

When Your Conduit Doesn't Appear in the Custom Dialog Box

If you do not see your conduit in HotSync Manager's **Custom** dialog box, it means that HotSync Manager has not loaded your conduit. Table 7.1 shows several possible reasons and solutions.

Table 7.1 Solutions for a conduit that is not loading

Reason	Solution(s)
Conduit built as an extension DLL	**NOTE:** You must build your conduit as a regular DLL. Do not build it as an extension DLL.
Incorrect path or conduit name	You must either specify a full path and filename or, if only a filename, the file must be in the HotSync Manager executable directory or in the Windows `PATH`. You can copy your conduit DLL to the HotSync directory and install your conduit by supplying the DLL name. See Chapter 3, "Conduit Configuration Utility," on page 11 in the *Conduit Development Utilities Guide*. Or, you can keep your conduit DLL in your own directory and supply the full path name of your DLL (in the `Conduit` configuration entry) when you install your conduit. In any case, check that the conduit path and filename are spelled correctly in the Conduit Configuration utility.

Table 7.1 Solutions for a conduit that is not loading *(continued)*

Reason	Solution(s)
Missing dependency	Any DLLs that your conduit links with must be available in the search path that HotSync Manager uses. This includes the `HotSync` directory, the Windows `System` directory, the `Windows` or `WinNT` directory, and the directories listed in the Windows `PATH` environment variable. Note that the Win32 call used to load a conduit is `LoadLibrary()`, which has a specific search order when loading libraries/dependant libraries that does *not* necessarily include the current location of the conduit (if a full path is specified). Refer to the Microsoft documentation on `LoadLibrary()`.
Missing function in dependency	Sometimes an older version of a required DLL is present, and the older version does not support all of the exported functions that a new one does. For example, the `sync20.dll` library. If you are using any of the new functions, your conduit cannot be loaded because the addresses for these imported functions cannot be resolved.

NOTE: You can use the Dependency Walker (`Depends.exe`) to discover dependency problems: www.dependencywalker.com.

When Your Conduit Does Not Run

Table 7.2 shows several possible reasons and solutions for a conduit that does not run, despite displaying in HotSync Manager's Custom dialog box.

Table 7.2 Solutions for a conduit that is not running

Reason	Solution(s)
Creator ID mismatch	HotSync Manager runs a conduit only if there is an application on the handheld with a matching creator ID, unless your conduit opts out of this requirement. See "GetConduitInfo()" on page 19.

Invalid version number	The version number of your conduit must be in the range specified by these two constants:
	`MIN_CONDUIT_VERSION 0x00000101`
	`MAX_CONDUIT_VERSION 0x00000200`
	The value that you return from your `GetConduitVersion()` entry point must be in the range `MIN_CONDUIT_VERSION` to `MAX_CONDUIT_VERSION`. Listing 3.2 shows an example of the `GetConduitVersion()` function.

Table 7.2 Solutions for a conduit that is not running

Reason	Solution(s)
Entry points	If you are using MFC in your conduit, you must include the `AFX_MANAGE_STATE(AfxGetStaticModuleState())` macro at the beginning of all of your entry points. You must explicitly set the current module state to the one for the DLL by using this macro at the beginning of every function that is exported from the DLL.
Incomplete class list	If your conduit uses classes that are implemented in another DLL, you must add that DLL to your `CDynLinkLibrary` chain. For example, if you are using the `CHotSyncActionDialog`, which is in the `pdcmnxx` DLL, you must call `InitPdcmdn5DLL` in your conduit's `InitInstance` function. Similarly, if you are using MFC OLE, MFC database (or DAO), or MFC Sockets support in your conduit DLL, you must call the predefined initialization function for each DLL that you use. See the MFC documentation for the `AfxDbInitModule()`, `AfxOleInitModule()`, `AfxNetInitModule()`, `AfxDaoInit()`, and `AfxDaoTerm()` functions.

8

Writing a Desktop Notifier

When it is possible for both an application on the desktop computer and a conduit to modify the same user data on the desktop at the same time, HotSync Manager can notify the desktop application when a HotSync operation is starting so that both are not changing data on the desktop at the same time. To enable this messaging, you must implement a **notifier**. A notifier is an optional DLL, entirely separate from a conduit, that you can implement to pass messages from HotSync Manager to your desktop application.

NOTE: If your conduit accesses desktop data but does not share that data with a desktop application, you do not need to provide a notifier.

For an overview of notifiers, see "Desktop Notifiers" on page 71, and for more on when HotSync Manager calls them, see "Calling Notifiers" on page 92 in *Introduction to Conduit Development*.

This chapter has the following sections:

Requirements

To have HotSync Manager call your notifier during a HotSync operation:

- Implement the Desktop Application Notification API specified in Chapter 18, "Desktop Application Notifier API," on page 1097 in the *C/C++ Sync Suite Reference*.
- Registered your notifier with HotSync Manager in one of two ways:
  - Use the Notifier Install Manager API, part of the Conduit Manager (`CondMgr.dll`). This API allows your installer application to programmatically register, modify, or unregister a notifier on an end user's computer. See Chapter 9, "Using the Notifier Install Manager API," on page 155.
  - Use the Conduit Configuration utility (`CondCfg.exe`), a Windows application that allows you to register and unregister notifiers on your development machine without writing code. See "Registering and Editing Notifier Information" on page 20 of the *Conduit Development Utilities Guide*.

Before calling any conduits and again after they complete, HotSync Manager calls each registered notifier with the following Desktop Notification API call:

```
BOOL HS_Notify (int nCode, int nUserId)
```

The *`nCode`* parameter value is a message code, and the *`nUserId`* parameter value is the ID of the user for whom the synchronization is being performed. The notifier must understand the *`nCode`* values that it receives and then tell the desktop application what it needs to know.

Table 8.1 describes the message codes that HotSync Manager sends to a notifier. The "Typical Response" column indicates how a desktop application should respond and what value the notifier should return to HotSync Manager.

IMPORTANT: The desktop application is not notified whether your individual conduit fails.

Table 8.1 HotSync Manager notification message codes

Message Code	Description	Typical Response
`HS_SYNC_FAILURE`	Notifies the desktop application that the synchronization has completed unsuccessfully.	The message does not indicate the reason for failure.
`HS_SYNC_QUERYOK`	Sent to desktop application to determine if it can be synchronized.	If the application can be synchronized, it returns `TRUE`. If the user is currently editing data and the application cannot be synchronized, it returns `FALSE` and displays a message to the user, instructing him or her to finish editing and retry the HotSync operation. HotSync Manager aborts synchronization operations.
`HS_SYNC_STARTED`	Notifies the desktop application of which user is about to be synchronized.	If the specified user is currently using the desktop application, the application saves the user's data to disk. If that save fails, the desktop application returns `FALSE` and HotSync Manager aborts synchronization operations. The desktop application returns `TRUE` in all other cases.

Table 8.1 HotSync Manager notification message codes *(continued)*

Message Code	Description	Typical Response
`HS_SYNC_SUCCESS`	Notifies the desktop application that synchronization has completed successfully.	If the list of users was modified, the application must reload its user data.
`HS_USER_UPDATE`	Notifies the desktop application that the user list has been modified.	The application must read in the list of users and reload user data after synchronization operations complete.

Example Notifier

The example notifier in Listing 8.1 is based on one that sends messages to Palm OS Desktop. The `HSNotify.h` file is located in `<CDK>\C++\Win\include`.

Listing 8.1 Example notifier

```
#include <windows.h>
#include <hsnotify.h>

const char* pszPilotDesktopWndClass = "Bell XS-1";
const char* pszHotSyncWndClass = "KittyHawk";

#define WM_POKEWILMA(WM_USER+0xBAC4)
#define PW_IDENTIFY0
#define PW_SYNCSTART1
#define PW_SYNCSUCCESS2
#define PW_SYNCFAILURE3
#define PW_ADVISEUSER4
#define PW_SYNCOKQUERY5
#define PW_UPDATEUSERS6

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD ul_reason_for_call,
                       LPVOID lpReserved )
```

```
{
   switch (ul_reason_for_call)
   {
      case DLL_PROCESS_ATTACH:
      case DLL_THREAD_ATTACH:
      case DLL_THREAD_DETACH:
      case DLL_PROCESS_DETACH:
      break;
   }
   return TRUE;
}

// GetNotifierVersion() is optional. HotSync Manager does not call it. However,
// it may be useful for your desktop application to call this function to
// identify the version of your notifier.
NOTIFY_API DWORD GetNotifierVersion()
{
   return 0;
}

// HS_Notify() passes messages from HotSync Manager to your
// desktop application.
NOTIFY_API BOOL HS_Notify(int nCode,int nUserID)
{
   BOOL bRtn = TRUE;

   // Find the Palm OS Desktop application.
   HWND hWnd = FindWindow(pszPilotDesktopWndClass,NULL);

   if (hWnd != NULL)
   {
      // Make sure it's the right Palm OS Desktop.
      if (SendMessage(hWnd,WM_POKEWILMA,PW_IDENTIFY,0) == WM_POKEWILMA)
      {
         switch (nCode)
         {
            case HS_SYNC_QUERYOK:
            bRtn = (SendMessage(hWnd,WM_POKEWILMA,PW_SYNCOKQUERY,0) == 0);
            break;

            case HS_SYNC_STARTED:
            bRtn = (SendMessage(hWnd,WM_POKEWILMA,PW_SYNCSTART,nUserID) == 0);
            break;

            case HS_SYNC_SUCCESS:
            SendMessage(hWnd,WM_POKEWILMA,PW_SYNCSUCCESS,nUserID);
            break;
```

```
            case HS_SYNC_FAILURE:
            SendMessage(hWnd,WM_POKEWILMA,PW_SYNCFAILURE,nUserID);
            break;

            case HS_USER_UPDATE:
            SendMessage(hWnd,WM_POKEWILMA,PW_UPDATEUSERS,nUserID);
            break;
         }
      }
   }

   return bRtn;
}
```

9

Writing an Installer

Creating an easy-to-use installer is an important part of product development. You need to make it easy for end users to install your conduit and your desktop and handheld applications. This chapter describes what your installer needs to do and the APIs that the C/C++ Sync Suite provides to help you create an installer.

The sections in this chapter are:

Installer Tasks

Your installer may need to install a handheld application and databases, a desktop application, as well as a conduit. So the best user experience is to write a single installer to install all of these at once. For handheld databases, your installer should display a user list if there is more than one HotSync® user and prepare the databases to be installed on the next HotSync operation. Your installer must register your conduit, otherwise HotSync Manager cannot run it. And finally, it must request that the user perform a HotSync operation to actually install your application and databases on the handheld.

To summarize, your installer must perform at least some of the following tasks to successfully deploy your product on an end user's desktop computer and handheld:

- Copy all of your install files to the end user's desktop computer.
- Install your desktop application, if your product includes one.
- Check that HotSync Manager is installed. Use the Conduit Manager API, `CondMgr.dll`. Required for any installer.
- Copy your conduit files and notifier DLL, if you provide one, to a final destination directory.
- Register your conduit with HotSync Manager. Use the Conduit Manager API, `CondMgr.dll`, either to register conventionally or to find the `Conduits` folder, if you are registering by folder. Registration is required for any conduit.
- Register your notifier with HotSync Manager. Use the Notifier Install Manager API, `CondMgr.dll`. Required only if you are providing a notifier.
- Restart HotSync Manager after you register your conduit. Use the HotSync Manager API, `HSAPI.dll`. Required only for a conduit registered with HotSync Manager versions earlier than 6.0. Versions 6.0 and later do not require this.
- Queue databases or applications to be installed on the handheld—or files to be installed on expansion cards—during the next HotSync operation. Use the Install Aide API,

`InstAide.dll`. Required for any databases or files to be put on the handheld.

- Uninstall your desktop application, conduit, and notifier, if you provide one—which includes unregistering your conduit and notifier with HotSync Manager.

Finding the HotSync Manager Binaries

One of the first tasks an installer must perform is to determine whether the HotSync Manager executable and its support DLLs are installed on an end user's computer. To do this, your installer must call Conduit Manager to get the path of the HotSync Manager executable. But the Conduit Manager library (`CondMgr.dll`) that your installer must use is in the same directory as HotSync Manager.

To break this circular dependency, PalmSource permits you to redistribute a temporary copy of `CondMgr.dll` in your installer for the sole purpose of finding the already-installed `CondMgr.dll` file on an end user's computer. After you find the installed copy, you must delete your temporary copy from the user's computer.

Unless otherwise specified, PalmSource does not permit you to redistribute any other binary files from the CDK.

WARNING! Always use the `CondMgr.dll` and other HotSync Manager support DLL files that are already installed on a user's computer. Do not replace the installed DLLs with other copies, which might be incompatible with the versions of other installed binaries. Mixing versions of HotSync Manager binaries on a user's computer can cause data loss.

The following steps outline how to determine whether HotSync Manager is already installed on a user's computer:

1. Include in your installer a copy of the `CondMgr.dll` file provided in the `<CDK>\Common\Bin` folder.
2. Copy `CondMgr.dll` to a temporary location on the end user's computer.

3. Using your temporary copy of `CondMgr.dll`, call `CmGetSystemHotSyncExecPath()`, if you're using Conduit Manager version 3 or later, or call `CmGetHotSyncExecPath()`, if you're using version 2 or earlier.

   If either of these functions passes back an empty string via the *`pPath`* parameter, then HotSync Manager is not installed. PalmSource recommends that you prompt the user to install the Palm OS® Desktop software that came with the handheld and then try to install your product again. Skip to step 6.

   However, if either of these functions passes back a nonempty string via the *`pPath`* parameter, then HotSync Manager is already installed. This parameter points to the full path and filename of the HotSync Manager executable. Proceed to the next step.

4. Strip the `HotSync.exe` filename from the path returned by `CmGetSystemHotSyncExecPath()`. This is the path of `CondMgr.dll` and other support DLL files that are already installed on the user's computer.

5. Using the installed `CondMgr.dll` file located in the path returned in step 4, call `CmGetLibVersion()` to get the version of the Conduit Manager API.

   This step confirms that you can successfully call the installed `CondMgr.dll` file and gives you the version number of its API. If your installer requires Conduit Manager functions that are available only in later versions, use this version number to determine whether and how to proceed. In "Conduit Manager Functions" on page 665 in the *C/C++ Sync Suite Reference*, the description of each function specifies which versions of Conduit Manager provide the function.

6. You *must* delete your temporary copy of `CondMgr.dll` (step 2) from the user's computer, whether HotSync Manager is already installed or not. PalmSource does not permit you to leave your temporary copy of `CondMgr.dll` on users' computers.

7. Call, as necessary, other installed HotSync Manager support DLLs that are in the same directory you discovered in step 4.

NOTE: To get the path of the HotSync Manager support DLLs, be sure to use `CmGetSystemHotSyncExecPath()`, not `CmGetCorePath()`. `CmGetCorePath()` retrieves the path of the HotSync users' folder, which for HotSync Manager versions earlier than 6.0, is the same as the HotSync executable path. However, in versions 6.0 and later these are two different paths.

Table 9.1 summarizes the Conduit Manager API functions described in this section.

Table 9.1 Finding HotSync Manager and Conduit Manager

Task	Conduit Manager API Functions
Get the path and filename of the HotSync Manager executable. Strip off the `HotSync.exe` filename to get the path of the folder that holds all HotSync Manager binaries.	`CmGetSystemHotSyncExecPath()`
Get the path of the HotSync users' directory for the current Windows user. Each Windows user must have a unique path, so there is no system-level path.	`CmGetCorePath()`

Files to Install

After it finds the installed HotSync Manager, your installer typically needs to install the following files on a user's computer:

- `MyApp.prc`, `MyAppData.pdb`. Your handheld application and possibly separate databases.
- `MyConduit.dll`. Your C API-based conduit.
- `MyApp.exe` and supporting files. Your desktop application, if your product includes one.
- `CondMgr.dll`, PalmSource's Conduit Manager library. You may copy this HotSync Manager support DLL to the user's computer only temporarily, as described in "Finding the HotSync Manager Binaries" on page 101.

Using the Conduit Manager API

This section describes the most common tasks an installer performs using the Conduit Manager:

HotSync Manager, versions 6.0 and later, and Conduit Manager, versions 3 and later, support multi-user versions of Windows. For background information, see "Support for Multiple Users" on page 64 in the *Introduction to Conduit Development*. The following

subsections refer to differences between conduits registered for the user and the system.

NOTE: Before calling Conduit Manager functions to change system-level settings, call `CmIsCurrentUserAdmin()` first to determine whether the current Windows user has administrator privileges. The system-level calls that make changes can succeed only if the user has these privileges; otherwise these calls return an `ERR_INSUFFICIENT_PRIVILEGES` and fail. System-level calls that only read information do not require administrator privileges.

Registering a Conduit Conventionally

This section describes how to use the Conduit Manager API to conventionally register a synchronization conduit. For background information and an explanation of terminology used in this section, see Chapter 6, "Registering Conduits and Notifiers with HotSync Manager," on page 73 in *Introduction to Conduit Development*.

The Conduit Manager API separates the underlying storage of conduit registration information from the processes of registering and unregistering conduits. This abstraction makes it possible for the underlying storage details to change without affecting how you register and unregister a conduit.

NOTE: Do not directly manipulate the Windows registry to register a conduit. If you circumvent the Conduit Manager to register a conduit, your installer may fail to work with future versions of HotSync Manager.

The Conduit Manager uses the unique creator ID assigned to each registered conduit to read and write values in the conduit's configuration entries. See Appendix A, "Configuration Entries," on page 175 in the *Introduction to Conduit Development* for descriptions of the conduit configuration entries that HotSync Manager reads.

You can use the Conduit Manager in two different ways to conventionally register your conduit, as described in the following subsections:

Writing a Single Structure

To register a C API-based conduit from a single structure, allocate a `CmConduitType2` structure, set the structure field values appropriately, and pass this structure to `CmInstallConduitByStruct()`. This function attempts to store your conduit's configuration entries for the creator ID you specified in the structure. If another conduit is already registered using the same creator ID, this function returns an

ERR_CREATORID_ALREADY_IN_USE error and does not register your conduit.

The only fields in CmConduitType2 that you *must* set to register a C API-based conduit are:

- dwCreatorID—the conduit's creator ID
- szConduitPath—the conduit DLL's path and filename (or only the filename)

The other fields are optional. See "CmConduitType2" on page 656 in the *C/C++ Sync Suite Reference* for details.

Listing 9.1 is an example of using CmInstallConduitByStruct() to register a conduit for the current Windows user. Simply replace this function call with CmInstallSystemConduitByStruct() to register the conduit for the system.

Listing 9.1 Example of registering a conduit with a single structure

```
int RegisterConduitByStruct () {
   CmConduitType2 pConduit;
   DWORD id;
   int retval = 0;

   memset(&pConduit, '\0', sizeof(pConduit));

   strcpy(pConduit.szConduitPath,"MyConduit.dll");
   strcpy(pConduit.szLocalDirectory, "MyConduitDir");
   strcpy(pConduit.szLocalFile,"MyDesktopFile.dat");
   strcpy(pConduit.szRemoteDB,"MyHandheldDatabase");
   strcpy(pConduit.szTitle,"My Conduit Display Name");

   CmConvertStringToCreatorID("MyID", &id);
   pConduit.dwCreatorID = id;
   pConduit.dwPriority = 2;

   retval = CmInstallConduitByStruct(pConduit);
// If retval is ERR_CREATORID_ALREADY_IN_USE, then another
// conduit is already registered with this creator ID, so
// this call fails.
   return retval;
}
```

Compatibility Note

The CmConduitType2 structure and CmInstallConduitByStruct() function are available in Conduit Manager API versions 3 and later. This structure contains only those configuration entries that all versions of HotSync Manager use and stores strings as simple character arrays.

All versions of the Conduit Manager provide the CmConduitType structure and CmInstallConduit() function to do the same thing. The `CmConduitType` structure differs from `CmConduitType2` in the following two ways:

- Several fields each specify the offset from the beginning of the structure to the beginning of a null-terminated string. This technique allows the structures to be built efficiently, without wasting string space. However, when you create a `CmConduitType` structure, you need to allocate the space for the structure itself and for the strings that you are specifying. If you wish to avoid this memory management work, PalmSource recommends that you use the `CmConduitType2` structure instead and pass it into the `CmInstallConduitByStruct()`.
- Several fields specify values that are deprecated in Conduit Manager versions 3 and later.

Table 9.2 summarizes the functions for registering a conduit with a structure for both the current Windows user and the system.

Table 9.2 Writing a single structure to conventionally register a conduit

Task	Conduit Manager API Functions
Register a conduit conventionally. Pass in a single CmConduitType2 structure. Available in Conduit Manager versions 3 and later.	User: CmInstallConduitByStruct() System: CmInstallSystemConduitByStruct()
Register a conduit conventionally. Pass in a handle to a single CmConduitType structure. Available in all versions of Conduit Manager. Note that there is no corresponding system-level function that uses this structure.	User: CmInstallConduit() System: N/A

Writing Individual Configuration Entries

To register a C API-based synchronization conduit by writing several configuration entries with a series of function calls, call CmInstallCreator() first with the creator ID of your conduit. If this function succeeds, then call the following series of functions, in no particular order, to set other conduit configuration values:

- CmSetCreatorName()—Besides CmInstallCreator(), this is the only other value you must set to register a conduit of any type. The rest are optional.
- CmSetCreatorPriority()
- CmSetCreatorTitle()
- CmSetCreatorDirectory()
- CmSetCreatorFile()
- CmSetCreatorRemote()

You can also create your own configuration entries and set them along with these at install time, as described in "Accessing Developer-defined Conduit Configuration Entries" on page 121.

Listing 9.2 shows a simple conduit registration that calls `CmInstallCreator()` and several `CmSetCreator...()` functions.

Listing 9.2 Registering a conduit with a series of Conduit Manager functions

```
int RegisterConduitWithMultipleCalls () {
   err = CmInstallCreator("Xyzz", CONDUIT_APPLICATION);
   if (err == 0)
      err = CmSetCreatorName("Xyzz",
         "C:\\MyCond\\Debug\\MyCond.DLL");
   if (err == 0)
      err = CmSetCreatorDirectory("Xyzz", "MyCond");
   if (err == 0)
      err = CmSetCreatorFile("Xyzz", "MyCond");
   if (err == 0)
      err = CmSetCreatorPriority("Xyzz", 2);
   if (err == 0)
      printf(" Registration succeeded\n");
   else
      printf(" Registration failed %d\n", err);
   return err;
}
```

Table 9.3 summarizes the functions for registering a conduit for both the current Windows user and the system by writing individual configuration entries.

Table 9.3 Writing individual configuration entries to conventionally register a conduit

Task	Conduit Manager API Functions

Register a conduit conventionally. Call `CmInstallCreator()` or `CmInstallSystemCreator()` first. Then call the rest, passing in the same creator ID.	User:
	`CmInstallCreator()`
	`CmSetCreatorName()`
	`CmSetCreatorPriority()`
	`CmSetCreatorTitle()`
	`CmSetCreatorDirectory()`
	`CmSetCreatorFile()`
	`CmSetCreatorRemote()`
	
	System:
	`CmInstallSystemCreator()`
	`CmSetSystemCreatorName()`
	`CmSetSystemCreatorPriority()`
	`CmSetSystemCreatorTitle()`
	`CmSetSystemCreatorDirectory()`
	`CmSetSystemCreatorFile()`
	`CmSetSystemCreatorRemote()`

For more information about these functions, see Chapter 11, "Conduit Manager API," on page 651 of the *C/C++ Sync Suite Reference*.

Registering a Conduit by Folder

This section describes how to use the Conduit Manager API to register a C API-based synchronization conduit by placing it in the system or current Windows user's `Conduits` folder. For background information and an explanation of terminology used in this section, see Chapter 6, "Registering Conduits and Notifiers with HotSync Manager," on page 73 in *Introduction to Conduit Development*.

NOTE: On an NTFS file system, moving any files, including conduits, in system-level areas can fail if the user does not have administrator privileges. You can call `CmIsCurrentUserAdmin()` first to determine whether the user has sufficient privileges.

A folder-registered conduit must implement logic in its `GetConduitInfo()` entry point to respond to an input *`infoType`* parameter value of `eRegistrationInfo`. See "GetConduitInfo()" on page 19 for more on this requirement.

To find the current Windows user's `Conduits` folder at install time, call `FmGetCurrentUserConduitFolder()`; for the system `Conduits` folder, call `FmGetSystemConduitFolder()`. These functions always return a path, because if a `Conduits` folder does not exist, these functions create them.

After you get the `Conduits` folder's path, copy your synchronization conduit DLL to that folder. The Conduit Manager finds your conduit there the next time HotSync Manager queries it for a list of conduits.

TIP: Most installers are designed to remove files during an uninstall from the location to which they were originally installed. Therefore implement this technique in your installer to keep your folder-registered conduit in the same location, at least as far as your installer is concerned:

1. Install your conduit to the `Disabled` subfolder.

2. Call `FmEnableCurrentUserConduitByPath()` or `FmEnableSystemConduitByPath()` to move it to the `Conduits` folder. Your installer thinks your conduit is still in the `Disabled` subfolder.

3. Upon uninstall, first call `FmDisableCurrentUserConduitByPath()` or `FmDisableSystemConduitByPath()` to move your conduit back to the `Disabled` subfolder. Then your installer can delete your conduit from its originally installed location.

This technique also covers the case of a power user who moves your conduit to the `Disabled` subfolder after installation. Upon uninstall, the call to disable the conduit fails, but your installer can still delete the conduit, because it is in its originally installed location.

Compatibility Note

Folder-based conduit registration is supported by versions 3 and later of the Conduit Manager (versions 6.0 and later of HotSync Manager). If you wish to install conduits on systems with earlier versions, you must register them conventionally as described in "Registering a Conduit Conventionally" on page 106.

Table 9.4 summarizes the functions for registering a conduit by folder for both the current Windows user and the system.

Table 9.4 Registering and unregistering a conduit by folder

Task	Conduit Manager API Functions

Find the `Conduits` folder and `Disabled` subfolder.	User:
	`FmGetCurrentUserConduitFolder()`
	`FmGetCurrentUserDisabledConduitFolder()`
	System:
	`FmGetSystemConduitFolder()`
	`FmGetSystemDisabledConduitFolder()`

Enable/disable a disabled/enabled conduit. Pass in the path and filename (or only filename) of the conduit.	User:
	`FmEnableCurrentUserConduitByPath()`
	`FmDisableCurrentUserConduitByPath()`
	System:
	`FmEnableSystemConduitByPath()`
	`FmDisableSystemConduitByPath()`

Disable an enabled conduit. Pass in the index of a folder-registered conduit.	User:
	`FmDisableCurrentUserConduitByIndex()`
	System:
	`FmDisableSystemConduitByIndex()`

Unregistering a Conduit

When you uninstall your conduit, you must unregister it so that HotSync Manager does not attempt to call your conduit DLL, which is no longer present. The following subsections describe how to unregister a conventionally registered and a folder-registered conduit.

Conventionally Registered

To unregister a synchronization conduit that is conventionally registered for the current Windows user, call `CmRemoveConduitByCreatorID()`, passing in the creator ID of the conduit to unregister. To do the same for a system-registered conduit, call `CmRemoveSystemConduitByCreatorID()`. These functions remove all of the conduit's configuration entries, but they do not delete the conduit DLL.

Table 9.5 summarizes the functions for unregistering a conventionally registered conduit for both the current Windows user and the system.

Table 9.5 Unregistering a conventionally registered conduit

Task	Conduit Manager API Functions
Unregister a conduit by *creator ID*.	User: `CmRemoveConduitByCreatorID()` System: `CmRemoveSystemConduitByCreatorID()`
Unregister a conduit by *index*. Call `CmGetConduitCount()` or `CmGetSystemConduitCount()` to get the upper limit of the index.	User: `CmRemoveConduitByIndex()` System: `CmRemoveSystemConduitByIndex()`

Folder-registered

You can unregister a folder-registered conduit in any of three ways:

- Delete your conduit DLL from the `Conduits` folder. To find the `Conduits` folder, call `FmGetCurrentUserConduitFolder()` or `FmGetSystemConduitFolder()`.
- Call `CmRemoveConduitByCreatorID()` or `CmRemoveSystemConduitByCreatorID()` and pass in the creator ID of the conduit to remove. The Conduit Manager moves the conduit from the `Conduits` folder to the `Disabled` subfolder.
- Call `CmRemoveConduitByIndex()` or `CmRemoveSystemConduitByIndex()` and pass in the index of the conduit to remove. The Conduit Manager moves the conduit from the `Conduits` folder to the `Disabled` subfolder.
- Call `FmDisableCurrentUserConduitByPath()` or `FmDisableSystemConduitByPath()` and pass in the path or filename of the conduit to remove. Or call `FmDisableCurrentUserConduitByIndex()` or `FmDisableSystemConduitByIndex()` and pass in the index of the conduit to remove. The Conduit Manager moves the conduit from the `Conduits` folder to the `Disabled` subfolder.

Moving a conduit from the `Conduits` folder to the `Disabled` subfolder effectively unregisters a conduit, so HotSync Manager does not attempt to call it. To re-register, or enable, a conduit that is in the `Disabled` subfolder, move it back to the `Conduits` folder by calling `FmEnableCurrentUserConduitByPath()` or `FmEnableSystemConduitByPath()` or by any other means.

NOTE: On an NTFS file system, moving any files, including conduits, in system-level areas can fail if the user does not have administrator privileges. You can call `CmIsCurrentUserAdmin()` first to determine whether the user has sufficient privileges.

See "Registering a Conduit by Folder" on page 112 for a tip on how to accommodate installers that must uninstall files from the same folder it installed them in.

Table 9.4 on page 114 provides a summary of the functions for unregistering a conduit by folder for both the current Windows user and the system.

Resolving Conduit Conflicts

As described in "Resolving Conduit Conflicts" on page 80 in the *Introduction to Conduit Development*, it is possible for two or more folder-registered conduits to have the same creator ID, or for one or more folder-registered conduit to have the same creator ID as a conventionally registered conduit. Before you register your conduit, you should confirm that no conduit is already registered with the same creator ID and thereby prevent potential conflicts.

Call `CmGetCreatorIDList()` to get a list of all user-registered creator IDs, if you are going to register your conduit only for the current Windows user; otherwise, call `CmGetSystemCreatorIDList()`. If your creator ID is already on the list, then prompt the user to choose whether to install your conduit or leave the existing one installed. If the user chooses to install your conduit, you must unregister the existing conduit before registering yours.

If you are conventionally registering your conduit, then Conduit Manager calls that do so return an `ERR_CREATORID_ALREADY_IN_USE` error and fail, if another conduit is already registered with the same creator ID. This is how Conduit Manager prevents conventionally registered conduits from conflicting. However, if you are registering your conduit by copying it to the `Conduits` folder, then the only way you can prevent a conflict is to check whether your creator ID is on the list of registered conduits, as described above.

Conduit Manager can warn you of an existing conflict in another way. It returns an `ERR_AMBIGUOUS_CREATORID` error and fails whenever you call a function that attempts to get or set a configuration entry for a registered conduit that already conflicts with another. Functions that can return this error are listed in Table 9.18 on page 143.

Accessing Registered Conduit Information

The Conduit Manager API provides functions for retrieving information about all registered synchronization conduits, *whether conventionally registered or folder-registered*. These APIs do not access install or backup conduits. Table 9.6 describes the kinds of information you can get or set with Conduit Manager APIs, for both user- and system-registered conduits.

Several APIs allow you to set the values of single conduit configuration entries in the same way you can if you conventionally register a conduit, as described in "Writing Individual Configuration Entries" on page 109.

NOTE: You can set conduit configuration entries only for conventionally registered conduits. If you call a function to create or set an entry for a folder-registered conduit, the function returns an `ERR_CONDUIT_READ_ONLY` error and fails.

Table 9.6 Accessing registered conduit information

Task	Conduit Manager API Functions
Get a count of all conduits. Use as upper limit of index in other by-index calls.	User: `CmGetConduitCount()` System: `CmGetSystemConduitCount()`
Get a list of all conduit creator IDs.	User: `CmGetCreatorIDList()` System: `CmGetSystemCreatorIDList()`
Get a conduit's creator ID by *index*.	User: `CmGetConduitCreatorID()` System: `CmGetSystemConduitCreatorID()`

Table 9.6 Accessing registered conduit information *(continued)*

Task	Conduit Manager API Functions
Get registration information by *index*. Passes back a `CmConduitType2` structure.	User: `CmGetConduitByIndex()` System: `CmGetSystemConduitByIndex()`
Determine by *index* whether a conduit is registered conventionally or is folder-based and whether HotSync Manager can load it. Passes back a `CmDiscoveryInfoType` structure.	User: `CmGetDiscoveryInfoByIndex()` System: `CmGetSystemDiscoveryInfoByIndex()`
Get conduit registration information by *creator ID*. Passes back a `CmConduitType` structure.	User: `CmGetConduitByCreator()` System: `CmGetSystemConduitByCreator()`

Table 9.6 Accessing registered conduit information *(continued)*

Task	Conduit Manager API Functions

Get standard conduit configuration entries by *creator ID*. These are the same entries as defined in the `CmConduitType2` structure.	User:
	`CmGetCreatorName()`
	`CmGetCreatorPriority()`
	`CmGetCreatorTitle()`
	`CmGetCreatorDirectory()`
	`CmGetCreatorFile()`
	`CmGetCreatorRemote()`
	
	System:
	`CmGetSystemCreatorName()`
	`CmGetSystemCreatorPriority()`
	`CmGetSystemCreatorTitle()`
	`CmGetSystemCreatorDirectory()`
	`CmGetSystemCreatorFile()`
	`CmGetSystemCreatorRemote()`

Set standard conduit configuration entries by *creator ID*. These are the same entries as defined in the `CmConduitType2` structure.	User:
	`CmSetCreatorName()`
	`CmSetCreatorPriority()`
	`CmSetCreatorTitle()`
	`CmSetCreatorDirectory()`
	`CmSetCreatorFile()`
	`CmSetCreatorRemote()`
	
	System:
	`CmSetSystemCreatorName()`
	`CmSetSystemCreatorPriority()`
	`CmSetSystemCreatorTitle()`
	`CmSetSystemCreatorDirectory()`
	`CmSetSystemCreatorFile()`
	`CmSetSystemCreatorRemote()`

Accessing Developer-defined Conduit Configuration Entries

The conduit **configuration entries** are extensible. You can define your own entries to store information per user-registered and per system-registered conduit. Table 9.7 describes the kinds of information you can get or set with Conduit Manager APIs. Your installer, conduit, or desktop application can store information in conduit configuration entries that are specific to your conduit.

When you unregister a conduit, the Conduit Manager removes all of a conduit's configuration entries, including those that you define. The Conduit Manager does not access your entries in any other circumstances.

NOTE: You can define your own conduit configuration entries only for conventionally registered conduits. If you call a function to create or set an entry for a folder-registered conduit, the function returns an `ERR_CONDUIT_READ_ONLY` error and fails.

Table 9.7 Accessing developer-defined conduit configuration entries

Task	Conduit Manager API Functions

Get or set an entry's `DWORD` value for a conduit by *creator ID*. The set function creates the entry if it does not already exist.	User:
	`CmGetCreatorValueDword()`
	`CmSetCreatorValueDword()`
	System:
	`CmGetSystemCreatorValueDword()`
	`CmSetSystemCreatorValueDword()`

Get or set an entry's string value for a conduit by *creator ID*. The set function creates the entry if it does not already exist.	User:
	`CmGetCreatorValueString()`
	`CmSetCreatorValueString()`
	System:
	`CmGetSystemCreatorValueString()`
	`CmSetSystemCreatorValueString()`

Retrieving Folder-registered Conduit Information

This section describes the functions that the Conduit Manager API provides for retrieving information about *folder-registered conduits only.* For functions that access information about both folder-registered and conventionally registered conduits, see "Accessing Registered Conduit Information" on page 118.

Table 9.8 describes the kinds of information you can get with Conduit Manager APIs, for both user- and system-registered conduits that are registered by folder.

NOTE: These functions only retrieve information. You cannot use the Conduit Manager API to set registration information for folder-registered conduits.

Table 9.8 Retrieving folder-registered conduit information

Task	Conduit Manager API Functions
Get a count of all folder-registered conduits. Use as upper limit of index in other by-index calls.	User: `FmGetCurrentUserConduitCount()` System: `FmGetSystemConduitCount()`
Get registration information by *index*. Passes back a `CmConduitType2` structure.	User: `FmGetCurrentUserConduitByIndex()` System: `FmGetSystemConduitByIndex()`

Registering and Unregistering a Backup Conduit

HotSync Manager runs only one backup conduit during a HotSync operation. (You can register one for the user and one of the system, but HotSync Manager calls only the user-registered backup conduit, if one is present; else it calls the system-registered backup conduit. See "User- and System-registered Conduits and Notifiers" on page 78 in *Introduction to Conduit Development*.) Therefore the Conduit Manager provides only functions to register/unregister and to get the filename of the backup conduit for the user and system, as described in Table 9.9.

WARNING! If you unregister the backup conduit, register another backup conduit to replace it; otherwise some of the user's data is not backed up during the next HotSync operation.

Table 9.9 Registering and unregistering the backup conduit

Task	Conduit Manager API Functions
Register a backup conduit by *filename*. To unregister, pass in the filename of a different backup conduit.	User: `CmSetBackupConduit()` System: `CmSetSystemBackupConduit()`
Get the filename of the registered backup conduit.	User: `CmGetBackupConduit()` System: `CmGetSystemBackupConduit()`

Configuring the COM Ports

The Conduit Manager provides functions for configuring the COM port that HotSync Manager uses for direct (local) and modem connections, as shown in Table 9.10. Note that there are no system-level COM port settings; these settings are for the current Windows user only. When each Windows user performs a HotSync operation for the first time, the Conduit Manager sets default COM port values for each user. You can use these functions to read or change these defaults, if you wish.

Table 9.10 Configuring the COM ports

Task	Conduit Manager API Functions
Get the name of the COM port that HotSync Manager uses for the specified type of connection.	User: `CmGetComPort()` System: N/A
Set the name of the COM port that HotSync Manager uses for the specified type of connection.	User: `CmSetComPort()` System: N/A

Using Conduit Manager's Utility Functions

The Conduit Manager API includes several utility functions summarized in Table 9.11.

Table 9.11 Using utility functions

Task	Conduit Manager API Functions
Convert a creator ID from a string/DWORD to a DWORD/string. Many Conduit Manager functions take a creator ID as a DWORD rather than as a string.	User: CmConvertCreatorIDToString() CmConvertStringToCreatorID() System: N/A
Get the version number of the Conduit Manager API. See the "Compatibility" section of each function to see in which versions of the Conduit Manager API the function is available.	User: CmGetLibVersion() System: N/A
Determine whether the current Windows user has administrator privileges. Call this function to determine whether any system-level Conduit Manager calls can succeed.	User: CmIsCurrentUserAdmin() System: N/A

Summary of Conduit Manager Functions

Conduit Manager Functions

Finding and Configuring HotSync Manager

CmGetCorePath()
CmGetComPort()
CmSetComPort()
CmGetPCIdentifier()
CmGetSystemHotSyncExecPath()

Registering/Unregistering Synchronization Conduits Conventionally

CmInstallCreator()
CmInstallConduitByStruct()
CmInstallSystemCreator()
CmInstallSystemConduitByStruct()

CmRemoveConduitByIndex()
CmRemoveConduitByCreatorID()
CmRemoveSystemConduitByIndex()
CmRemoveSystemConduitByCreatorID()

CmInstallConduit()

Registering/Unregistering Folder-registered Conduits

FmGetCurrentUserConduitFolder()
FmGetCurrentUserDisabledConduitFolder()
FmDisableCurrentUserConduitByPath()
FmDisableCurrentUserConduitByIndex()
FmEnableCurrentUserConduitByPath()
FmGetSystemConduitFolder()
FmGetSystemDisabledConduitFolder()
FmDisableSystemConduitByPath()
FmDisableSystemConduitByIndex()
FmEnableSystemConduitByPath()

Reading Registered Conduit Information

CmGetConduitCount()
CmGetCreatorIDList()
CmGetConduitCreatorID()
CmGetConduitByCreator()
CmGetConduitByIndex()
CmGetDiscoveryInfoByIndex()
CmGetSystemConduitCount()
CmGetSystemCreatorIDList()
CmGetSystemConduitCreatorID()
CmGetSystemConduitByCreator()
CmGetSystemConduitByIndex()
CmGetSystemDiscoveryInfoByIndex()

Conduit Manager Functions *(continued)*

Accessing Standard Conduit Configuration Information

CmGetCreatorName()
CmSetCreatorName()
CmGetCreatorPriority()
CmSetCreatorPriority()
CmGetCreatorTitle()
CmSetCreatorTitle()
CmGetCreatorDirectory()
CmSetCreatorDirectory()
CmGetCreatorFile()
CmSetCreatorFile()
CmGetCreatorRemote()
CmSetCreatorRemote()

CmGetSystemCreatorName()
CmSetSystemCreatorName()
CmGetSystemCreatorPriority()
CmSetSystemCreatorPriority()
CmGetSystemCreatorTitle()
CmSetSystemCreatorTitle()
CmGetSystemCreatorDirectory()
CmSetSystemCreatorDirectory()
CmGetSystemCreatorFile()
CmSetSystemCreatorFile()
CmGetSystemCreatorRemote()
CmSetSystemCreatorRemote()

Accessing Developer-defined Conduit Configuration Entries

CmGetCreatorValueDword()
CmSetCreatorValueDword()
CmGetCreatorValueString()
CmSetCreatorValueString()

CmGetSystemCreatorValueDword()
CmSetSystemCreatorValueDword()
CmGetSystemCreatorValueString()
CmSetSystemCreatorValueString()

Retrieving Folder-registered Conduit Information

FmGetCurrentUserConduitCount()
FmGetCurrentUserConduitByIndex()

FmGetSystemConduitCount()
FmGetSystemConduitByIndex()

Registering and Unregistering a Backup Conduit

CmGetBackupConduit()
CmSetBackupConduit()

CmGetSystemBackupConduit()
CmSetSystemBackupConduit()

Configuring the COM Ports

CmGetComPort()
CmSetComPort()

Utility Functions

CmConvertCreatorIDToString()
CmConvertStringToCreatorID()

CmGetLibVersion()
CmIsCurrentUserAdmin()

Using the Install Aide API

The Install Aide enables your desktop application or installer to queue files for installation on a handheld during a HotSync operation. This section describes how the Install Aide works and the most common tasks that an installer performs using the Install Aide:

For details on the Install Aide functions, see Chapter 15, "Install Aide API," on page 951 in the *C/C++ Sync Suite Reference*.

How Install Aide Works

The Install Aide does not actually move data between the desktop and handheld; it simply moves files into a queue directory on the desktop and tells HotSync Manager that something needs to be installed. A registered **install conduit** called by HotSync Manager during a HotSync operation actually transfers queued files to primary storage or to an expansion card in a handheld. HotSync Manager ships with several install conduits, so you likely do not need to implement your own install conduit. "Running Install Conduits" on page 105 in the *Introduction to Conduit Development* lists these install conduits and the file types they can install.

The Install Aide API provides two functions that your installer, desktop application, or conduit can call to queue files for installation: `PltInstallFile()` for files destined for primary storage and `PlmSlotInstallFile()` for files destined for an expansion card. When you call one of these functions, you specify the HotSync user and the path of the file you want to install. At the

time you call one of these functions, Install Aide performs the following tasks:

1. Install Aide selects a default install conduit to run based on the *extension* of the file you specify and the install *function* you call, as shown in Table 9.12. Each install conduit is registered to handle certain filename extensions.

Table 9.12 Install Aide's choice of install conduit and destination based on filename extension

Filename Extension	Function Called	Install Conduit used by Install Aide	Destination on Handheld
PRC, PDB, SDB, SSD, PQA	`PltInstallFile()`	Install	Primary storage
PNC, SCP	`PltInstallFile()`	Install Service Templates[1]	Primary storage
.	`PltInstallFile()`	HotSync Exchange	Primary storage
.	`PlmSlotInstallFile()`	Install to Card	Card in specified expansion slot

1. Works only with handhelds running a version of Palm OS earlier than Palm OS Cobalt.

   Install Aide chooses the HotSync Exchange conduit if the filename extension does not match one of the preceding Palm OS filename extensions but does match an extension that a handheld application has registered with the Exchange Manager on the handheld. See Chapter 3, "Using HotSync Exchange," on page 27 in the *Introduction to Conduit Development*.

NOTE: The files you install with `PltInstallFile()` must have filename extensions. Otherwise, the install function fails and returns `ERR_IA_INVALID_FILE_TYPE`, because Install Aide cannot determine what install conduit to use.

2. Install Aide copies the file to the install directory that is associated with the install conduit that it selected in step 1.
3. Install Aide creates an install flag for the specified HotSync user in the configuration entries.
4. Install Aide sets the value of the install flag to the value of the bit mask associated with the install conduit that is registered to handle files of the specified type. HotSync Manager reads this install flag the next time it runs install conduits for the specified user. This flag tells HotSync Manager that it must run the install conduit with the matching mask value.

If you call one of the install functions again before HotSync Manager runs the install conduits again, these steps repeat—except that the Install Aide ORs the existing value of the install flag with that of each subsequent call. For example, if the first call selects an install conduit with a bit mask value of 1 and the second call selects one with a value of 4, the resulting value of this install flag is 5 the next time the install conduits are run. Therefore HotSync Manager will run two install conduits, one registered with a bit mask of 1 and another with a bit mask of 4.

Each install conduit decides whether to remove its mask value from the HotSync user's install flag after a successful or unsuccessful installation. The Install install conduit leaves the mask value if unsuccessful; it removes it if successful. The HotSync Exchange install conduit runs independent of the mask, because there could be files that need to be transferred from the handheld to the desktop.

HotSync Manager calls install conduits twice: before it runs all synchronization conduits and again after. If one of the install functions is called outside of a HotSync operation, then an install conduit installs the queued file at the *beginning* of the *next* HotSync operation. If a conduit calls one of the install functions during a HotSync operation, then an install conduit installs the queued file at the *end* of the *current* HotSync operation.

The Install Aide provides other functions to help manage files queued to be installed and to obtain expansion slot information. These are discussed in the remainder of this section and detailed in Chapter 15, "Install Aide API," on page 951 in the *C/C++ Sync Suite Reference*.

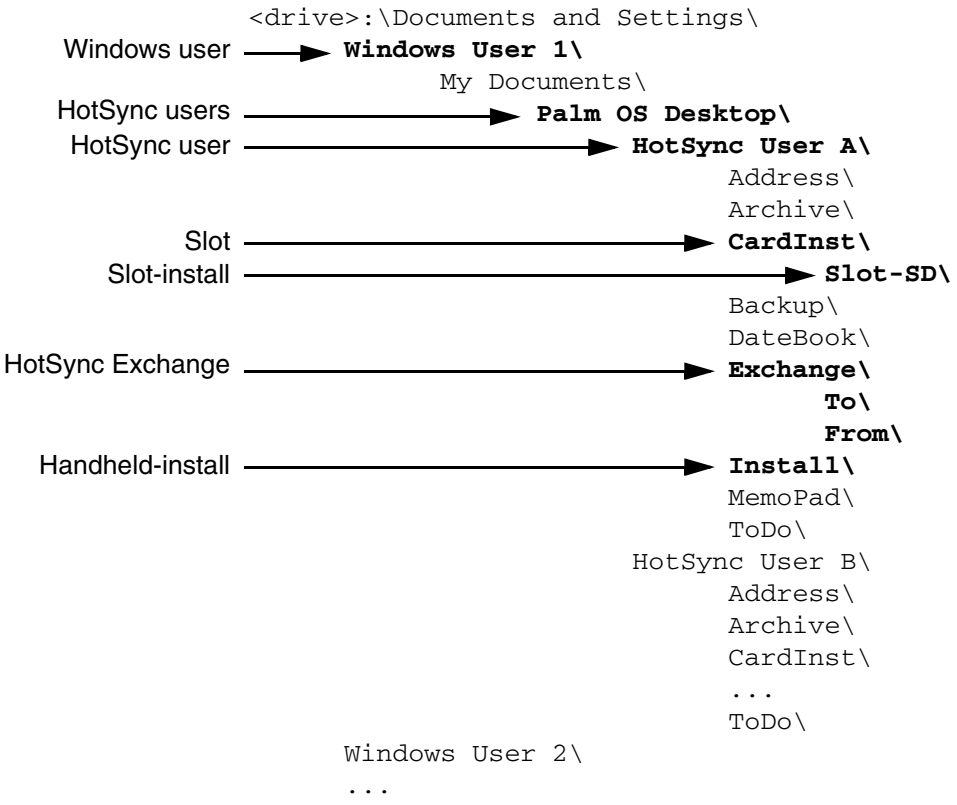
Install Directory Terminology

Table 9.13 defines the various desktop directories to which the Install Aide copies files queued to be installed by the default install conduits during the next HotSync operation. Figure 9.1 shows the relative locations of these directories for the current Windows user.

Table 9.13 Desktop directories associated with Install Aide

Directory Name	Description
Windows user's directory	For the current Windows user, this directory contains all HotSync Manager files and user directories (by default) for all HotSync users. To get this path, call `CmGetCorePath()`.
HotSync user's directory	For each HotSync user, this directory contains the handheld-install and slot directories, as well as data directories for each default conduit.
Handheld-install directory	For each HotSync user, this directory contains Palm OS applications and databases queued to be installed by an associated install conduit into the handheld's main memory the next time install conduits run. There can be such a directory for each of multiple install conduits.
Slot directory	For each HotSync user, this directory contains subdirectories (slot-install directories) for each slot on the user's handheld. This directory and its subdirectories are associated with an install conduit that installs files to expansion cards.
Slot-install directory	For each slot on each HotSync user's handheld, this directory contains files queued to be installed to a card in the associated slot the next time install conduits run.
HotSync Exchange directory	For each HotSync user, the `Exchange` directory contains `To` and `From` subdirectories. The `To` directory contains files to be installed to the handheld's main memory via HotSync Exchange. The `From` directory contains files that have been sent to the desktop from the handheld.

Figure 9.1 Directories that default install conduits queue files in



Specifying File Types

The `PltGetFileCount()`, `PltGetFileInfo()`, and `PltGetFileName()` functions require that the caller pass in the filename in the *`pExtension`* parameter. This parameter requires a string of format *"*.extension"*—for example, the following are the standard Palm OS database image filename extensions:

- `"*.prc"`—Application
- `"*.pdb"`—Classic or extended database
- `"*.sdb"`—Schema database
- `"*.ssd"`—Secure schema database
- `"*.pqa"`—Query application
- `"*.pnc"`—Network configuration file
- `"*.scp"`—Network script file

If you implement an install conduit, you can register your own filename extensions when you register your install conduit. See "Using the Install Conduit Manager API" on page 158.

Those functions that pass back a file extension do so in the same format—for example, `PltGetFileTypeExtension()`.

The functions that pass back file filters, `PltGetInstallFileFilter()` and `PltGetInstallFileFilterForUser()`, do so in a buffer containing a string that specifies the file filter types concatenated together in Windows-standard format—for example:

```
"type1 (*.ext1)|*.ext1|type2 (*.ext2)|*.ext2||"
```

Queuing a Database to Install in Primary Storage

The Install Aide provides functions that enable you to queue a Palm OS database image file to be installed in the handheld's primary storage memory the next time HotSync Manager runs install conduits. To perform this operation, pass the HotSync user's name and the path of the file to the `PltInstallFile()` function. The Install Aide does the rest, as described in "How Install Aide Works" on page 128.

The default install conduit named Install is registered to handle only files with the extensions of Palm OS database image files. For differences on how to use Install Aide to queue files of other types for installation in primary storage, see "Queuing a File to Install in Primary Storage via HotSync Exchange" on page 136.

Listing 9.3 is an example of using `PltInstallFile()` to queue an application (`MyApp.prc`) for installation in primary storage on the handheld.

Listing 9.3 Example of queuing a file to install in primary storage

```
int QueueFileToInstallInPrimary () {

int iRetval = 0;
TCHAR pUser[45];
short psUserBufSize = sizeof (pUser);
unsigned int iIndex = 0;

   // Get the HotSync user's name. This example just gets the
   // first user in the user data store.
   psUserBufSize = sizeof (pUser);
   memset (pUser, 0, psUserBufSize);
   iRetval = PltGetUser (iIndex, pUser, &psUserBufSize);

   if (iRetval > 0)
   {
      // Queue file for installation for the first user.
      iRetval = PltInstallFile (pUser, "C:\MyApp.prc");
   }
   return iRetVal;
}
```

Table 9.14 summarizes the functions for queuing a file to install in primary storage.

Table 9.14 Queuing a database to install in primary storage

Task	Install Aide API Functions
Queue a file to install in primary storage for a HotSync user specified by *name*. File must be of a type supported by a registered install conduit.	`PltInstallFile()`
Remove a queued file from the queue folder for a HotSync user specified by *name*.	`PltRemoveInstallFile()`
Move a file that is queued for installation on an expansion card to be installed in primary storage instead. Specify the HotSync user by *user ID*.	`PlmMoveInstallFileToHandheld()`

Get the number, names, and sizes of files queued for installation in primary storage. Specify the HotSync user by *name*.	`PltGetFileCount()`
	`PltGetFileInfo()`
	`PltGetFileName()`

Get the number and extensions of all file types supported by registered install conduits that install files in a handheld's primary storage.	`PltGetFileTypeExtension()` `PltGetFileTypesCount()`

Queuing a File to Install in Primary Storage via HotSync Exchange

Queuing a *file* to install in primary storage is similar to queuing an image file of a Palm OS *database*. See "Queuing a Database to Install in Primary Storage" on page 134 for how that is done. When you queue a *file* for installation in *primary storage* and it is not a Palm OS database image file, the Install Aide uses the *HotSync Exchange* install conduit. The primary differences in how this works are:

- Only versions 6.0 and later of HotSync Manager include the HotSync Exchange install conduit. Only Palm OS Cobalt has the required HotSync Exchange Manager library used by the Exchange Manager on the handheld.
- The default install conduit that Install Aide uses is called "HotSync Exchange." It is registered to handle only those file types that handheld applications have registered with the Exchange Manager.
- HotSync Manager retrieves information from the HotSync Exchange Manager library (or the lack thereof) at the beginning of each HotSync operation and saves it for the corresponding user in the user data store on the desktop. However, between HotSync operations, the user can change or update the handheld—for example, the user can add or delete an application that registers with the Exchange Manager for certain filename extensions. Upon the next HotSync operation, queued files with extensions that a now-deleted application registered for cannot be handled by the HotSync Exchange install conduit, so they are not installed.

For background on HotSync Exchange, see Chapter 3, "Using HotSync Exchange," on page 27 in the *Introduction to Conduit Development*.

Queuing a File to Install on an Expansion Card

Queuing a file to install on an expansion card is similar to queuing a file to install in primary storage, so see "Queuing a Database to Install in Primary Storage" on page 134. The primary differences are:

- Only versions 4.0 and later of the Install Aide API support expansion cards.
- Call `PlmSlotInstallFile()` to queue a file to install on a card.
- The default install conduit for expansion cards is called "Install to Card" and is registered to handle all filename extensions.
- HotSync Manager retrieves information about a handheld's expansion slots (or lack thereof) at the beginning of each HotSync operation and saves it for the corresponding user in the user information store on the desktop. However, between HotSync operations, the user can change or update the handheld—for example, the user can upgrade to a handheld with more slots or switch to one with no slots at all. Upon the next HotSync operation, HotSync Manager asks the user either to create a new HotSync user name or to choose one from the list of users who have previously performed a HotSync operation as the current Windows user. If the user chooses a HotSync user name from the list, the slot information saved for that user at the beginning of the last HotSync operation is now inaccurate.
- If your application or installer calls the Install Aide outside of a HotSync operation, be aware that your files might not be installed during the next HotSync operation if the slot or card does not exist at that time.
- Call the User Data API to get information about the expansion slots and cards in users' handhelds. You must pass user IDs and expansion slot IDs to expansion-related Install Aide APIs. See "Using the User Data API" on page 145.

Listing 9.4 is an example of using `PlmSlotInstallFile()` to queue a file (`MyPhoto.jpg`) for installation on an expansion card.

Listing 9.4 Example of queuing a file to install on an expansion card

```
long QueueFileToInstallOnCard () {

long kSuccess = 0;
short sIndex = 0;                       // Index of the first HotSync user.
long retval = 0;
DWORD dwUserID = 0;
WORD numSlots = 0;
DWORD *pdwSlotIdList;
char szFile[] = "\\MyPhoto.jpg"; // Name of file to queue for installation.

   // Get the HotSync user's name. This example just gets the first user in the
   // user data store.
   retval = UmGetUserID (sIndex, &dwUserID);
   if (retval == kSuccess)
   {
      // Get the number of expansion slots.
      retval = UmSlotGetSlotCount (dwUserID, &numSlots);
      if (retval == kSuccess)
      {
         if(numSlots > 0 )
         {
            // Get the slot IDs.
            pdwSlotIdList = new DWORD[numSlots];
            retval = UmSlotGetInfo (dwUserID, pdwSlotIdList, &numSlots);
            if (retval == kSuccess)
            {
               // Queue file to install on card in the first expansion slot.
               retval = PlmSlotInstallFile (dwUserID, *pdwSlotIdList, szFile);
               if (retval == kSuccess)
               {
                  delete[] pdwSlotIdList;
                  return retval;
               }
            }
         }
         else
         {
            return retval; // User's handheld has no slots.
         }
      }
   }
}
```

Table 9.15 summarizes the Install Aide functions for queuing a file to install in primary storage.

Table 9.15 Queuing a file to install on an expansion card

Task	Install Aide API Functions
Queue a file to install on an expansion card. Specify the HotSync user by *user ID* and the expansion slot by *slot ID*.	`PlmSlotInstallFile()`
Remove a queued file from the slot-install folder. Specify the HotSync user by *user ID* and the expansion slot by *slot ID*.	`PlmSlotRemoveInstallFile()`
Move a file that is queued for installation in primary storage to be installed on an expansion card instead. Specify the HotSync user by *user ID* and the expansion slot by *slot ID*.	`PlmMoveInstallFileToSlot()`
Move a file that is queued for installation on an expansion card to be installed on another expansion card instead. Specify the HotSync user by *user ID* and the expansion slots by *slot ID*.	`PlmSlotMoveInstallFile()`
Get the number, names, and sizes of files queued for installation on an expansion card. Specify the HotSync user by *user ID* and the expansion slot by *slot ID*.	`PlmSlotGetFileCount()` `PlmSlotGetFileInfo()`

Retrieving HotSync User Information

The Install Aide API provides the functions listed in Table 9.16 for retrieving HotSync user information from the user data store on the desktop. However, PalmSource, Inc. recommends that you use the User Data API instead. See "Using the User Data API" on page 145 for details.

Table 9.16 Retrieving HotSync user information

Task	Install Aide API Functions
Get a HotSync user name by *index*. Call `PltGetUserCount()` to get the upper limit of the index.	`PltGetUser()` `PltGetUserCount()`
Get a HotSync user's folder name. Specify the HotSync *user name*.	`PltGetUserDirectory()`
Get a HotSync user name/ID by specifying a *user ID/name*.	`PlmGetUserIDFromName()` `PlmGetUserNameFromID()`
Determine whether a HotSync user is actually a **user profile**.	`PltIsUserProfile()`

Accessing Registered Install Conduit Information

The Install Aide API provides functions for retrieving information about the install conduits that HotSync Manager can run for the current Windows user. (The Install Aide accesses only those install conduits that are on the Conduit Manager's list of reconciled install conduits. For more information see "User- and System-registered Conduits and Notifiers" on page 78 in the *Introduction to Conduit Development*. To get information about all registered install conduits, use the Install Conduit Manager discussed in "Using the Install Conduit Manager API" on page 158.) These APIs do not access synchronization or backup conduits. Table 9.17 describes the kinds of install conduit information you can get or set with Install Aide APIs.

Table 9.17 Accessing registered install conduit information

Task	Install Aide API Functions
Get a count of all registered install conduits. Use as upper limit of index in other by-index calls.	`PltGetInstallConduitCount()`
Get install conduit registration information by *index*. Passes back a `FileInstallType` structure.	`PltGetInstallConduitInfo()`
Get install conduit registration information by *unique ID*. Passes back a `FileInstallType` structure.	`PltGetInstallCreatorInfo()`

Table 9.17 Accessing registered install conduit information *(continued)*

Task	Install Aide API Functions
Get a file filter consisting of all filename extensions registered by all install conduits for all HotSync users, or for the specified HotSync user. When you specify the HotSync user, Install Aide takes into account whether the specified user's handheld has been synchronized with an expansion card in its slot; it includes related file type filters only if a card has been present before.	`PltGetInstallFileFilter()` `PltGetInstallFileFilterForUser()`

Determine by *index* whether a conduit is registered conventionally or is folder-based and whether HotSync Manager can load it. Passes back a `CmDiscoveryInfoType` structure.	User:
	`CmGetDiscoveryInfoByIndex()`
	System:
	`CmGetSystemDiscoveryInfoByIndex()`

Determine whether an install conduit is set to run or clear its install flag so that it does not run.	`PltIsInstallMaskSet()` `PltResetInstallMask()`

Using Install Aide's Utility Functions

The Install Aide API includes several utility functions summarized in Table 9.11.

Table 9.18 Using the Install Aide's utility functions

Task	Install Aide API Functions
Get the version number of the Install Aide API. See the "Compatibility" section of each function to see in which versions of the Install Aide API the function is available.	`PlmGetLibVersion()`
Get one of the path of the HotSync users' folder for the current Windows user. Each Windows user must have a unique path, so there is no system-level value.	`PltGetPath()`

Summary of Install Aide Functions

Install Aide Functions	
Queuing a Database or a File to Install in Primary Storage	
PltInstallFile()	PltGetFileCount()
PltRemoveInstallFile()	PltGetFileInfo()
PlmMoveInstallFileToHandheld()	PltGetFileName()
	PltGetFileTypeExtension()
	PltGetFileTypesCount()
Queuing a File to Install on an Expansion Card	
PlmSlotInstallFile()	PlmSlotGetFileCount()
PlmSlotRemoveInstallFile()	PlmSlotGetFileInfo()
PlmMoveInstallFileToSlot()	
PlmSlotMoveInstallFile()	
Retrieving HotSync User Information	
PltGetUser()	PlmGetUserIDFromName()
PltGetUserCount()	PlmGetUserNameFromID()
PltGetUserDirectory()	
PltIsUserProfile()	
Accessing Registered Install Conduit Information	
PltGetInstallConduitCount()	PltIsInstallMaskSet()
PltGetInstallConduitInfo()	PltResetInstallMask()
PltGetInstallCreatorInfo()	
PltGetInstallFileFilter()	
PltGetInstallFileFilterForUser()	
Utility Functions	
PlmGetLibVersion()	PltGetPath()

Using the User Data API

The User Data API is the preferred way to access information in the HotSync **user data store** on the desktop. (Other APIs offer some similar functionality, but PalmSource recommends that you use the User Data API instead.) The users data store holds the name, ID, synchronization preferences, desktop directory, and password for each HotSync user created by the current Windows user.

This section describes the most common tasks that an installer or desktop application performs using the User Data API:

For details on the User Data API functions, see Chapter 16, "User Data API," on page 1009 in the *C/C++ Sync Suite Reference*.

Adding and Deleting HotSync Users

If your desktop application or installer needs to manage HotSync users on the desktop, the User Data API provides the functions described in Table 9.19.

Table 9.19 Adding and deleting HotSync users

Task	User Data API Functions
Add a new HotSync user or **user profile** to the user data store. Specify the *user name*. If the user data store does not exist, this function creates it.	`UmAddUser()`
Delete a HotSync user. Specify the *user ID*.	`UmDeleteUser()`

Finding and Setting the Directory of a HotSync User

When HotSync Manager is installed, the **default conduits** store their desktop data in subdirectories of each HotSync user's directory as shown in Figure 9.1 on page 132. If you want to do the same, Table 9.20 describes the User Data API functions you can use to find or set this directory for the current Windows user.

Table 9.20 Finding and setting the directory of a HotSync user

Task	User Data API Functions
Get the path of the HotSync users' directory for the current Windows user. This is the value stored in the `Core\Path` configuration entry.	`UmGetRootDirectory()`
Get or set a HotSync user's directory name. Specify the HotSync *user ID*.	`UmGetUserDirectory()` `UmSetUserDirectory()`

Accessing Information about a HotSync User

The User Data API provides functions for accessing information about each HotSync user created by the current Windows user. Table 9.21 describes how to access this information.

Table 9.21 Accessing information about a HotSync user

Task	User Data API Functions
Get a HotSync user ID by *index*. Call `UmGetUserCount()` to get the upper limit of the index.	`UmGetUserID()` `UmGetUserCount()`
Get or set a HotSync user's name by *user ID*.	`UmGetUserName()` `UmSetUserName()`
Get a HotSync user's encrypted password by *user ID*. Call `PwdVerify()` to verify that the user's input matches.	`UmGetUserPassword()`
Determine whether a HotSync user has performed at least one HotSync operation as the current Windows user (is the "installed" flag set). Or set the "installed" flag. Specify a HotSync user's name by *user ID*.	`UmIsUserInstalled()` `UmSetUserInstall()`
Determine whether a HotSync user is actually a **user profile**. Specify a HotSync user's name by *user ID*.	`UmIsUserProfile()`

Accessing the Synchronization Preferences of a HotSync User

The User Data API provides functions for accessing a HotSync user's permanent and temporary synchronization preferences. These are the preferences pass between HotSync Manager and your conduit via its `CfgConduit()` entry point so that your conduit can present them to the user to change. For background on synchronization preferences, see "User's Conduit Synchronization Preferences" on page 62 in the *Introduction to Conduit Development*.

Table 9.22 describes how to access this information with User Data API functions.

Table 9.22 Accessing the synchronization preferences of a HotSync user

Task	User Data API Functions

Get or set a permanent or temporary synchronization preference of a conduit for a HotSync user specified by *user ID*.	`UmGetUserPermSyncPreferences()`
	`UmSetUserPermSyncPreferences()`
	`UmGetUserTempSyncPreferences()`
	`UmSetUserTempSyncPreferences()`

Delete the permanent or temporary synchronization preferences of *all* conduits for a HotSync user specified by *user ID*.	`UmDeleteUserPermSyncPreferences()` `UmDeleteUserTempSyncPreferences()`
Delete the temporary synchronization preference of a conduit for a HotSync user specified by *user ID*.	`UmRemoveUserTempSyncPreferences()`

Retrieving the Expansion Slot Information of a HotSync User

At the beginning of the HotSync process, HotSync Manager retrieves from the handheld whether it has expansion slots, and if so, information about cards in those slots. HotSync Manager stores that information in the user data store, so that it can be accessed between HotSync operations—for example, the Install Tool uses it to determine whether a HotSync user can queue files to be installed during the next HotSync operation. Because the User Data API does not retrieve slot information directly from the handheld but only what is saved in the user data store during the last HotSync operation, this information may not be accurate for the next HotSync operation; the user may have changed or updated the handheld in the interim.

Table 9.23 describes how to access this expansion slot information with User Data API functions.

Table 9.23 Retrieving the expansion slot information of a HotSync user

Task	User Data API Functions
Get the version of the Expansion Manager library on the handheld of a HotSync user specified by *user ID*.	`UmSlotGetExpMgrVersion()`
Get the number of expansion slots on the handheld of a HotSync user specified by *user ID*.	`UmSlotGetSlotCount()`
Get the slot IDs of all the expansion slots on the handheld for the HotSync user specified by *user ID*.	`UmSlotGetInfo()`
Get the display name of the expansion slot specified by *slot ID* for the HotSync user specified by *user ID*.	`UmSlotGetDisplayName()`

Table 9.23 Retrieving the expansion slot information of a HotSync user *(continued)*

Task	User Data API Functions
Get the type of media in a slot specified by *slot ID* for the HotSync user specified by *user ID*.	`UmSlotGetMediaType()`
Get the name of the slot-install directory on the desktop for a slot specified by *slot ID* for the HotSync user specified by *user ID*.	`UmSlotGetInstallDirectory()`

Modifying the Install Conduit Flags of a HotSync User

Some functions for accessing a user's install conduit flags perform the same action in the User Data API as some do in the Install Aide API. However, the User Data API adds `UmSetInstallMask()`. For more information, see "How Install Aide Works" on page 128 and "Accessing Registered Install Conduit Information" on page 141.

Table 9.24 describes how to modify and get the state of a user's install conduit flags.

Table 9.24 Modifying the install conduit flags of a HotSync user

Task	User Data API Functions
Determine whether an install conduit specified by its *mask* value is set to run during the next HotSync operation for the HotSync user specified by *user ID*.	`UmIsInstallMaskSet()`
Select or deselect an install conduit specified by its *mask* value so that it set to run or not to run during the next HotSync operation for the HotSync user specified by *user ID*.	`UmSetInstallMask()` `UmClearInstallMask()`

Accessing Developer-defined Entries in the User Data Store

In the same way that the Conduit Manager API enables you to define your own conduit configuration entries, the User Data API does the same for the user data store. But whereas the conduit configuration entries are stored per conduit for the current Window user, information in the user data store is stored per HotSync user.

The user data store is divided into named sections containing named keys whose values can be set. You can create sections, keys, and set and get the values of keys.

Table 9.25 describes the functions you can call to get and set integer and string values in the user data store.

Table 9.25 Accessing developer-defined entries in the user data store

Task	User Data API Functions

Get or set an entry's integer or string value for a HotSync user specified by *user ID*. The set function creates the section and key if they do not already exist.	`UmGetInteger()`
	`UmSetInteger()`
	`UmGetString()`
	`UmSetString()`

Delete a key or entire section for a HotSync user specified by *user ID*.	`UmDeleteKey()`

Using User Data API Utility Functions

The User Data API includes several utility functions summarized in Table 9.26.

Table 9.26 Using User Data API utility functions

Task	User Data API Functions
Get the version number of the User Data API. See the "Compatibility" section of each function to see in which versions of the User Data API the function is available.	`UmGetLibVersion()`
Get a HotSync user ID by specifying a HotSync user's *name* or *directory*.	`UmGetIDFromName()` `UmGetIDFromPath()`

Summary of User Data API Functions

User Data API Functions

Adding and Deleting HotSync Users

UmAddUser()
UmDeleteUser()

Finding and Setting a HotSync User's Directory

UmGetUserDirectory()
UmGetRootDirectory()
UmSetUserDirectory()

Accessing HotSync User Information

UmGetUserCount()
UmGetUserID()
UmGetUserName()
UmSetUserName()
UmGetUserPassword()
UmIsUserInstalled()
UmSetUserInstall()
UmIsUserProfile()

Accessing HotSync User Synchronization Preferences

UmGetUserPermSyncPreferences()
UmSetUserPermSyncPreferences()
UmDeleteUserPermSyncPreferences()
UmGetUserTempSyncPreferences()
UmSetUserTempSyncPreferences()
UmDeleteUserTempSyncPreferences()
UmRemoveUserTempSyncPreferences()

Retrieving HotSync User's Expansion Slot Information

UmSlotGetDisplayName()
UmSlotGetExpMgrVersion()
UmSlotGetInfo()
UmSlotGetInstallDirectory()
UmSlotGetMediaType()
UmSlotGetSlotCount()

Modifying Install Conduit Flags

UmClearInstallMask()
UmSetInstallMask()
UmIsInstallMaskSet()

Accessing Developer-defined Entries in the User Data Store

UmGetInteger()
UmSetInteger()
UmGetString()
UmSetString()
UmDeleteKey()

User Data API Functions *(continued)*	
Utility Functions	
`UmGetIDFromName()`	`UmGetLibVersion()`
`UmGetIDFromPath()`	

Using the HotSync Manager API

The HotSync Manager API enables your desktop application or installer to control the HotSync Manager application as summarized in Table 9.27.

Before calling any other HotSync Manager API functions, call the `HsCheckApiStatus()` function to determine whether you can communicate with HotSync Manager.

Table 9.27 Using the HotSync Manager API

Task	HotSync Manager API Functions
Determine whether the HotSync Manager API can communicate with the HotSync Manager application.	`HsCheckApiStatus()`
Determine whether a HotSync operation is in progress.	`HsGetSyncStatus()`
Start, restart, or exit the HotSync Manager executable.[1]	`HsSetAppStatus()`
Refresh HotSync Manager. This function requests HotSync Manager to reload its list of registered conduits.[1]	`HsRefreshConduitInfo()`

Display a HotSync Manager dialog box to customize conduits or change connection settings; or to display the HotSync Log.	`HsDisplayCustomDlg()`
	`HsDisplayLog()`
	`HsDisplaySetupDlg()`

Get or set the state (enabled or disabled) of a HotSync Manager connection type (serial, USB, network, and so on).	`HsGetCommStatus()` `HsSetCommStatus()`

Table 9.27 Using the HotSync Manager API *(continued)*

Task	HotSync Manager API Functions
Re-initialize all enabled HotSync Manager communications transports.	HsResetComm()
Get the version number of the HotSync Manager API. See the "Compatibility" section of each function to see in which versions of the HotSync Manager API the function is available. Note that this is *not* the version number of the HotSync Manager application.	HsGetApiVersion()

1. HotSync Manager versions *earlier than 6.0* must be refreshed or restarted after registering a conduit. HotSync Manager, versions *6.0 and later,* automatically refreshes its list so that this call is unnecessary.

For details on the HotSync Manager API functions, see Chapter 14, "HotSync Manager API," on page 927 in the *C/C++ Sync Suite Reference*.

Summary of HotSync Manager API Functions

HotSync Manager API Functions

Checking, Starting, Stopping and Refreshing HotSync Manager

HsSetAppStatus()
HsCheckApiStatus()
HsGetSyncStatus()
HsRefreshConduitInfo()

Displaying HotSync Manager Dialog Boxes

HsDisplayCustomDlg()
HsDisplaySetupDlg()
HsDisplayLog()

Getting and Setting Communications State

HsGetCommStatus()
HsSetCommStatus()
HsResetComm()

Utility Functions

HsGetApiVersion()

Using the Notifier Install Manager API

The Notifier Install Manager registers or unregisters **notifier**s with HotSync Manager. Its API separates the underlying storage of notifier registration information from the processes of registering and unregistering notifiers. This abstraction makes it possible for the underlying storage details to change without impact on how you install or uninstall your notifier.

The Notifier Install Manager uses only the filename of your notifier to register it with HotSync Manager. Alternatively, you can specify a fully qualified path, but note that these functions access a notifier only by the exact string you supply when you register it. Therefore if you specify a full path when you register a notifier, you must specify the same full path when you unregister it.

This section describes the most common tasks an installer performs using the Notifier Install Manager:

See Chapter 13, "Notifier Install Manager API," on page 909 in the *C/C++ Sync Suite Reference* for details on these functions.

Registering and Unregistering a Notifier

HotSync Manager runs all registered notifiers with unique paths and filenames only at the beginning and end of the HotSync process. Table 9.28 summarizes the functions for registering and unregistering notifiers for the current Windows user and for the system.

Table 9.28 Registering and unregistering a notifier

Task	Notifier Install Manager API Functions

Register a notifier by *filename*.	User:
	`NmRegister()`
	System:
	`NmRegisterSystem()`

Unregister a notifier by *filename*.	User:
	`NmUnregister()`
	System:
	`NmUnregisterSystem()`

Accessing Registered Notifier Information

The Notifier Install Manager API provides separate functions for retrieving information about notifiers registered for the current Windows user and for the system. Table 9.29 summarizes these functions.

Table 9.29 Accessing registered notifier information

Task	Notifier Install Manager API Functions
Get a count of all registered notifiers. Use as upper limit of index in other by-index calls.	User: NmGetCount() System: NmGetSystemCount()
Get a notifier's filename by *index*.	User: NmGetByIndex() System: NmGetSystemByIndex()
Change the filename of a registered notifier by *index*.	User: NmRenameByIndex() System: NmRenameSystemByIndex()
Get the index of a registered notifier by *filename*.	User: NmFind() System: NmFindSystem()

Summary of Notifier Install Manager Functions

Notifier Install Manager Functions	
Registering/Unregistering Notifiers	
NmRegister()	NmRegisterSystem()
NmUnregister()	NmUnregisterSystem()
Accessing Registered Notifier Information	
NmFind()	NmFindSystem()
NmGetCount()	NmGetSystemCount()
NmGetByIndex()	NmGetSystemByIndex()
NmRenameByIndex()	NmRenameSystemByIndex()

Using the Install Conduit Manager API

The Install Conduit Manager provides most of the same functionality for registering and unregistering install conduits as the Conduit Manager provides for registering standard synchronization conduits. An **install conduit** is a special type of C API-based conduit that installs Palm OS databases and applications on a handheld (or copies files from the desktop to handheld expansion cards). For more information on how the Install Aide uses install conduits, see "How Install Aide Works" on page 128.

NOTE: Most conduits are synchronization conduits, not install conduits, and must be registered with the Conduit Manager instead (see "Using the Conduit Manager API" on page 104).

This section describes the most common tasks an installer performs using the Install Conduit Manager:

Registering an Install Conduit

This section describes how to use the Install Conduit Manager API to register an install conduit. For background information and an explanation of terminology used in this section, see Chapter 6, "Registering Conduits and Notifiers with HotSync Manager," on page 73 in *Introduction to Conduit Development*.

The Install Conduit Manager API separates the underlying storage of install conduit registration information from the processes of registering and unregistering install conduits. This abstraction makes it possible for the underlying storage details to change without affecting how you register and unregister an install conduit.

NOTE: Do not directly manipulate the Windows registry to register an install conduit. If you do not use the Install Conduit Manager for install conduit registration, your installer may fail to work with future versions of HotSync Manager.

When you register an install conduit with HotSync Manager, you must specify a unique bit mask value (`Mask`), a unique ID (`CreatorID`), and one or more filename extensions (`Extensions`) associated with your install conduit. HotSync Manager uses only the mask value to identify your install conduit, but the Install Conduit Manager requires the unique ID to access and store values its configuration entries. This ID is not necessarily related to the creator ID you register with PalmSource, Inc. However, one way to ensure your value is unique is to use a creator ID that you registered with PalmSource, Inc. You use the Install Conduit Manager API to add these and other values associated with your install conduit in the install conduit configuration entries.

NOTE: The Install Conduit Manager checks only your install conduit's `CreatorID` against those of install conduits that are already registered. Because your `Mask` value must also be unique, you must first check it against those of install conduits that are already registered. The Install Conduit Manager returns an error when your `CreatorID` value is the same, but not when your `Mask` value is the same. Therefore your installer must ensure the uniqueness of the `Mask` value.

See "Install Conduit Configuration Entries" on page 184 in the *C/C++ Sync Suite Reference* for descriptions of the predefined install conduit configuration entries.

You can use the Install Conduit Manager in two different ways to register your install conduit, as described in the following subsections:

Writing a Single Structure

To register an install conduit from a single structure, allocate a `FileInstallType` structure and the strings associated with that structure, set the structure field values appropriately, and pass this structure to `ImRegister()`. This function attempts to store your install conduit's configuration entries for the unique ID you specified in the structure. If another install conduit is already registered with the same unique ID or filename extensions, this function returns an error and does not register your install conduit.

The only fields in `FileInstallType` that you *must* set to register an install conduit are:

- `dwCreatorID`—the install conduit's unique ID
- `szExt`—the filename extensions of files that an install conduit can install
- `szModule`—the install conduit DLL's path and filename (or only the filename)
- `dwMask`—the install conduit's unique bit mask

The other fields are optional. See "FileInstallType" on page 848 in the *C/C++ Sync Suite Reference* for details.

Listing 9.5 is an example of using `ImRegister()` to register a conduit for the current Windows user. Simply replace this function call with `ImRegisterSystem()` to register an install conduit for the system.

Listing 9.5 Example of registering an install conduit with a single structure

```
void RegisterInstallConduitByStruct () {
   FileInstallType pInstConduit;
   DWORD id;
   int retval = 0;

   memset(&pInstConduit, '\0', sizeof(pInstConduit));

   strcpy(pInstConduit.szModule,"MyInstConduit.dll");
   strcpy(pInstConduit.szDir, "MyInstallDir");
   strcpy(pInstConduit.szName,"My Install Conduit Display
      Name");
   strcpy(pInstConduit.szExt,"*.ext");

   CmConvertStringToCreatorID("MyID", &id);
   pInstConduit.dwCreatorID = id;
// Check the mask value of all registered install conduits
// and use a value that is not already in use.
   pInstConduit.dwMask = 0x00000001;

   retval = ImRegister(pInstConduit);
// If retval is not 0, then registration fails. Check for
// negative error codes.
}
```

Table 9.30 summarizes the functions for registering a conduit with a structure for both the current Windows user and the system.

Table 9.30 Writing a single structure to register an install conduit

Task	Install Conduit Manager API Functions

Register an install conduit. Pass in a single `FileInstallType` structure.	User:
	`ImRegister()`
	
	System:
	`ImRegisterSystem()`

Writing Individual Configuration Entries

To register an install conduit by writing several configuration entries with a series of function calls, call `ImRegisterID()` first with the unique ID of your install conduit. If the `ImRegisterID()` function succeeds, then call the following series of functions, in no particular order, to set other install conduit configuration values:

- `ImSetExtension()`—required
- `ImSetModule()`—required
- `ImSetMask()`—required
- `ImSetDirectory()`—optional
- `ImSetName()`—optional

You can also create your own configuration entries and set them along with these at install time, as described in "Accessing Developer-defined Install Conduit Configuration Entries" on page 166.

Listing 9.6 shows a simple install conduit registration that calls `ImRegisterID()` and several `ImSet...()` functions.

Listing 9.6 Registering an install conduit with a series of Install Conduit Manager functions

```
int RegisterInstallConduitWithMultipleCalls () {
   DWORD id;
   CmConvertStringToCreatorID("MyID", &id);

   err = ImRegisterID(id);
   if (err == 0)
      err = ImSetExtension()(id, "*.ext");
   if (err == 0)
      err = ImSetModule()(id,
         "C:\\MyCond\\Debug\\MyInstCond.DLL");
   if (err == 0)
      // Check the mask value of all registered install
      // conduits and use a value that is not already in use.
      err = ImSetMask()(id, 0x00000001);
   if (err == 0)
      err = ImSetDirectory()(id, "MyInstallDir");
   if (err == 0)
      err = ImSetName()(id, "My Install Conduit Display
         Name");
```

```
   if (err == 0)
      printf(" Registration succeeded\n");
   else
      printf(" Registration failed %d\n", err);
   return err;
}
```

Table 9.31 summarizes the functions for registering an install conduit for both the current Windows user and the system by writing individual configuration entries.

Table 9.31 Writing individual configuration entries to register an install conduit

Task	Install Conduit Manager API Functions

Register an install conduit. Call `ImRegisterID()` or `ImRegisterSystemID()` first. Then call the rest, passing in the same unique ID.	User:
	`ImRegisterID()`
	`ImSetExtension()`
	`ImSetModule()`
	`ImSetMask()`
	`ImSetDirectory()`
	`ImSetName()`
	
	System:
	`ImRegisterSystemID()`
	`ImSetSystemExtension()`
	`ImSetSystemModule()`
	`ImSetSystemMask()`
	`ImSetSystemDirectory()`
	`ImSetSystemName()`

For details on these functions, see Chapter 12, "Install Conduit Manager API," on page 847 in the *C/C++ Sync Suite Reference*.

Unregistering an Install Conduit

When you uninstall your install conduit, you must unregister it so that HotSync Manager does not attempt to call your install conduit DLL, which is no longer present. To unregister an install conduit, call `ImUnregisterID()`, passing in the unique ID of the install conduit to unregister. To do the same for a system-registered conduit, call `ImUnregisterSystemID()`. These functions remove all of the install conduit's configuration entries, but they do not delete the install conduit DLL.

Table 9.32 summarizes the functions for unregistering a registered install conduit for both the current Windows user and the system.

Table 9.32 Unregistering a registered install conduit

Task	Install Conduit Manager API Functions
Unregister an install conduit by *creator ID*.	User: `ImUnregisterID()` System: `ImUnregisterSystemID()`

For details on these functions, see Chapter 12, "Install Conduit Manager API," on page 847 in the *C/C++ Sync Suite Reference*.

Accessing Information about All Registered Install Conduits

The Install Conduit Manager API provides functions for retrieving information about all registered install conduits. These APIs do not access synchronization conduits or backup conduits. Table 9.33 describes the kinds of information you can get or set with Install Conduit Manager APIs, for both user- and system-registered conduits.

Several APIs allow you to set the values of single install conduit configuration entries in the same way you can if you register an install conduit as described in "Writing Individual Configuration Entries" on page 162.

To iterate through a list of only the install conduits that HotSync Manager can run for current Windows user, use the Install Aide

functions described in "Accessing Registered Install Conduit Information" on page 141.

Table 9.33 Accessing registered install conduit information

Task	Install Conduit Manager API Functions

Get standard install conduit configuration entries by *unique ID*. These are the same entries as defined in the `FileInstallType` structure.	User:
	`ImGetExtension()`
	`ImGetModule()`
	`ImGetMask()`
	`ImGetDirectory()`
	`ImGetName()`
	
	System:
	`ImGetSystemExtension()`
	`ImGetSystemModule()`
	`ImGetSystemMask()`
	`ImGetSystemDirectory()`
	`ImGetSystemName()`

Set standard install conduit configuration entries by *unique ID*. These are the same entries as defined in the `FileInstallType` structure.	User:
	`ImSetExtension()`
	`ImSetModule()`
	`ImSetMask()`
	`ImSetDirectory()`
	`ImSetName()`
	
	System:
	`ImSetSystemExtension()`
	`ImSetSystemModule()`
	`ImSetSystemMask()`
	`ImSetSystemDirectory()`
	`ImSetSystemName()`

For details on these functions, see "Install Conduit Manager Functions" on page 850 in the *C/C++ Sync Suite Reference*.

Accessing Developer-defined Install Conduit Configuration Entries

The install conduit **configuration entries** are extensible. You can define your own entries by unique ID to store information per user-registered and per system-registered install conduit. Table 9.34 describes the kinds of information you can get or set with the Install Conduit Manager APIs. Your installer, install conduit, or desktop application can store information in install conduit configuration entries that are specific to your install conduit.

When you unregister an install conduit, the Install Conduit Manager removes all of an install conduit's configuration entries, including those that you define. Otherwise, the Install Conduit Manager does not access your entries in any way. For this reason, you can call these functions at any time without having to refresh or restart HotSync Manager.

Table 9.34 Accessing developer-defined install conduit configuration entries

Task	Install Conduit Manager API Functions

Get or set an entry's `DWORD` value for an install conduit by *unique ID*. The set function creates the entry if it does not already exist.	User:
	`ImGetDWord()`
	`ImSetDWord()`
	System:
	`ImGetSystemDWord()`
	`ImSetSystemDWord()`

Get or set an entry's string value for an install conduit by *unique ID*. The set function creates the entry if it does not already exist.	User:
	`ImGetString()`
	`ImSetString()`
	System:
	`ImGetSystemString()`
	`ImSetSystemString()`

For details on these functions, see "Install Conduit Manager Functions" on page 850 in the *C/C++ Sync Suite Reference*.

Summary of Install Conduit Manager Functions

Install Conduit Manager Functions

Registering/Unregistering Install Conduits

ImRegister()	ImRegisterSystem()
ImUnregisterID()	ImUnregisterSystemID()

Accessing Standard Install Conduit Configuration Information

ImGetExtension()	ImGetSystemExtension()
ImSetExtension()	ImSetSystemExtension()
ImGetModule()	ImGetSystemModule()
ImSetModule()	ImSetSystemModule()
ImGetMask()	ImGetSystemMask()
ImSetMask()	ImSetSystemMask()
ImGetDirectory()	ImGetSystemDirectory()
ImSetDirectory()	ImSetSystemDirectory()
ImGetName()	ImGetSystemName()
ImSetName()	ImSetSystemName()

Accessing Developer-defined Install Conduit Configuration Entries

ImGetDWord()	ImGetSystemDWord()
ImSetDWord()	ImSetSystemDWord()
ImGetString()	ImGetSystemString()
ImSetString()	ImSetSystemString()

Uninstalling Your Conduit

Users expect that applications provide them with a facility to easily remove all the relevant files and other information that were put on the their computer during installation. Your uninstall application must:

- Unregister your conduit. See "Unregistering a Conduit" on page 115.
- Unregister your notifier DLLs, if you installed them. See "Registering and Unregistering a Notifier" on page 156.
- Delete any entries in the user data store, if you created any. See "Accessing Developer-defined Entries in the User Data Store" on page 151.
- Remove all installed files.
- Perform any other cleanup.

And, if your conduit replaced a conduit that was already installed on the user's machine (like any of the default conduits), you should offer the user the option to reregister it with HotSync Manager, leaving the user's machine exactly as you found it.

Testing Your Installer

Because one of the most important goals of the HotSync process is simplicity for users, ensure that your conduit installs and uninstalls correctly and easily. PalmSource, Inc. strongly recommends that, at a minimum, you test the following conduit installation and removal conditions:

- Check for the existence of HotSync Manager on the user's desktop computer. See "Finding the HotSync Manager Binaries" on page 101.
- After installing your conduit, the user can press the HotSync button and have synchronization operations proceed correctly and without required intervention.
- After removing (uninstalling) your conduit, the user can press the HotSync button and have synchronization operations proceed correctly and without required intervention.

- After removal of your conduit, the conduit configuration entries are restored to what they were prior to its installation.

NOTE: Whenever you change any of the configuration entries, HotSync Manager versions earlier than 6.0 require that you call either `HsSetAppStatus()` to restart HotSync Manager to recognize all but conduit configuration changes or `HsRefreshConduitInfo()` to recognize a conduit configuration change.

However, HotSync Manager versions 6.0 and later automatically discover changes to conduit configuration information without you calling these functions. To recognize other changes (HotSync Manager communication settings, backup conduit, and so on), you must still call `HsSetAppStatus()` to restart HotSync Manager.

In addition to verifying that your conduit installation and removal operations leave the user's desktop computer in fully operational condition, you need to be aware of the following situations on the handheld that can cause your conduit to generate unpleasant side effects for users:

- If your conduit stores state information on the handheld, you must remember to clean up the state information upon removal of your conduit.
- If you create data for a creator ID that you do not own, your conduit can generate very unpleasant results. Do not use a creator ID that does not belong to you.

Installation Troubleshooting Tips

This section suggests a few things to look for when you have installation problems:

- Check the HotSync log after you run your conduit. The log contains any error messages that may result. Ensure that you are looking at the log entry for the correct user.

- For additional (verbose) logging information, exit HotSync Manager, click the **Start** button, then **Run**, and type `c:\<HotSync Manager dir>\hotsync.exe -v`.
- Conduits are run by HotSync Manager only when there is a application with a matching creator ID on the handheld, unless your conduit tells HotSync Manager to run it regardless. See "GetConduitInfo()" on page 19.
- Use `CondCfg.exe` to view the conduit configuration settings. Double check them and watch for things like whether your conduit DLL is actually where you indicated.
- Ensure that the creator ID specified for the conduit (use `CondCfg.exe` to view the entry) matches the creator ID of an application on the handheld.
- Use the verbose log to get a list of the creator IDs on the handheld.

See the *Conduit Development Utilities Guide* for details on using the Conduit Configuration utility (`CondCfg.exe`) and other helpful tools.

Sample Installer

For a sample conduit installer, see the article "Sample Code for a Conduit Installer in C++" in the Palm OS Knowledge Base. This article provides a Microsoft Visual C++ 6.0 project that generates a conduit installer.

A

Quick Start: Using Visual C++ .NET to Build a Conduit

This quick-start guide steps you through the process of creating and starting to debug a sample conduit generated by the Conduit Wizard in Visual C++ .NET. The conduit you will create synchronizes the Memo Port database on the handheld with an XML file on the desktop. The Memo Port application is a version of the Palm OS® Cobalt Memo Pad application that uses an extended database rather than a schema database. For details on the differences between these types of databases, see "Schema vs. Non-schema Databases" on page 114 in *Introduction to Conduit Development*.

The code that the wizard generates in this quick-start guide is based on the Extended Generic Conduit Framework, which works with **extended database**s. When you are finished with this guide, you can examine the generated code as an example of how to use this framework in your own conduit.

This guide is divided into the following major steps:

- Step 1: Use the Conduit Wizard to Create Sample Code
- Step 2: Prepare Palm OS Cobalt Simulator
- Step 3: Configure the Project Settings and Build the DLL
- Step 4: Single-step/Debug Your Conduit
- Step 5: Examine the MyConduitData.xml File

This guide assumes:

- You installed and are using Visual C++ .NET (hereafter referred to as VC).

- You installed the CDK's C/C++ Sync Suite in the default location (<CDK>).
- You installed the Palm OS Cobalt Simulator. To get Simulator, see Chapter 8, "Synchronizing with Palm OS Simulator," on page 49 in the *Conduit Development Utilities Guide*.

NOTE: Because a part of the CDK installation process includes plugging the Conduit Wizard into VC, install VC *before* you install the CDK. Otherwise the Conduit Wizard will not be available. If VC does not show **Palm OS® Conduit** in the **New** > **Projects** > **Visual C++ Projects** dialog box, you need to install the Conduit Wizard manually. See "Installing the Conduit Wizard" on page 4 in *Conduit Development Utilities Guide*.

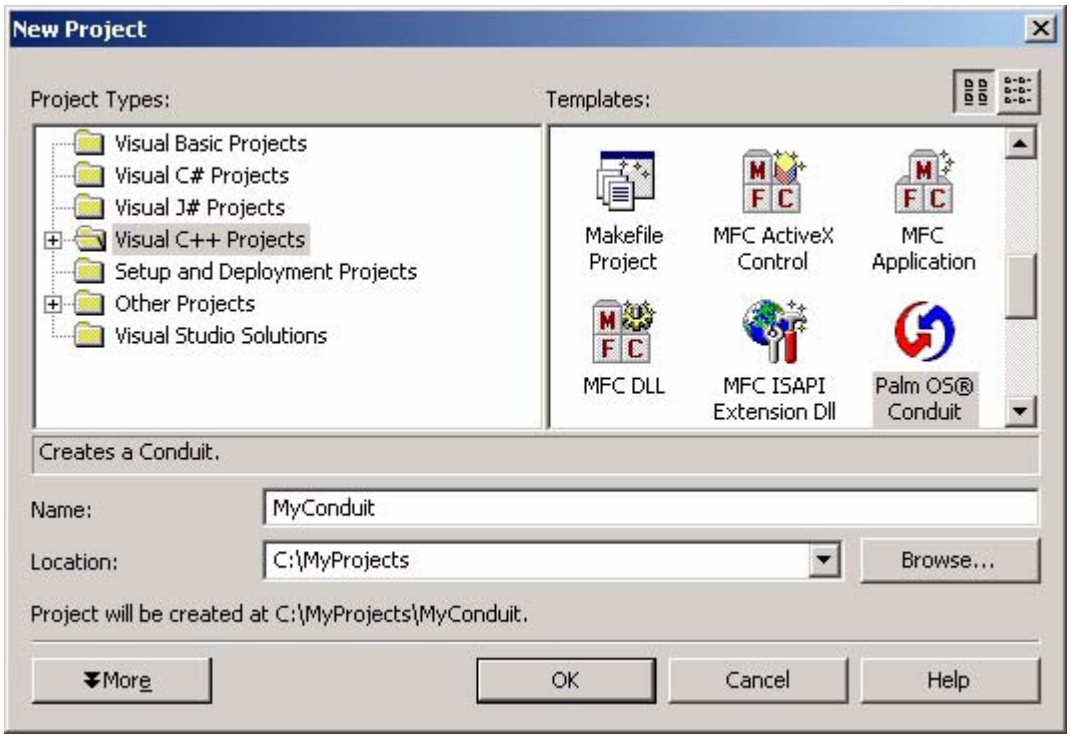
Step 1: Use the Conduit Wizard to Create Sample Code

In this step, you will use the Conduit Wizard to create conduit code that will synchronize the database used by the Memo Port application on the handheld with an XML file on the desktop.

Start the Conduit Wizard

1. Start VC and click **File** > **New** > **Project**.
2. Under **Project Types**, click the **Visual C++ Projects** folder to display all the installed C++ project wizards as shown in Figure A.1.

Figure A.1 Selecting the Conduit Wizard



3. Under **Templates**, click **Palm OS® Conduit**.
4. In the **Name** box, enter `MyConduit` as the name for your project and enter a location for it.

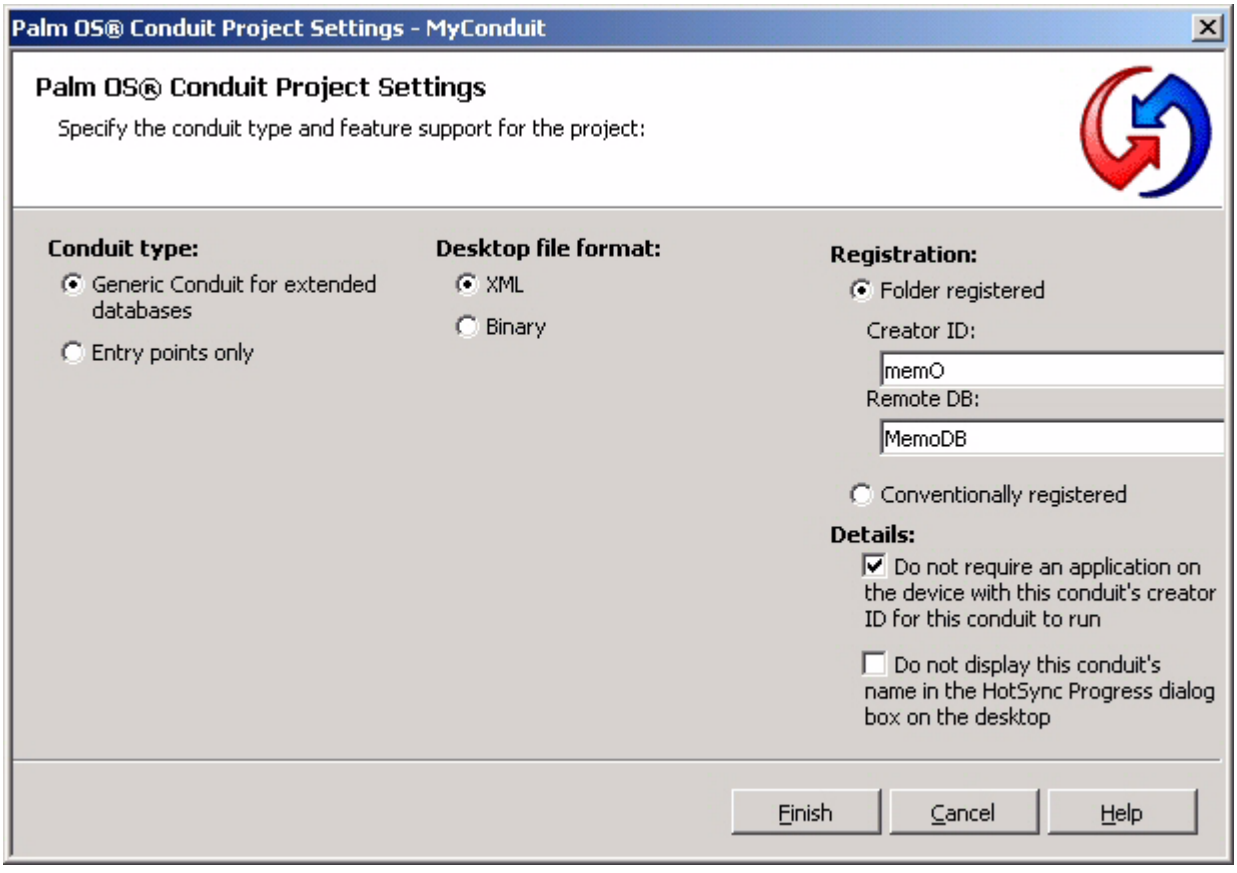
IMPORTANT: Choose your project name carefully, because the Conduit Wizard uses it in filenames and object names. If your project name does not follow the rules for C identifiers, then your project might not compile.

5. Click **OK**.

Use the Conduit Wizard

The Conduit Wizard then presents you with choices to make for the design of your conduit (Figure A.2). Each choice is described in the steps that follow.

Figure A.2 Conduit Wizard choices



6. Under **Conduit type**, click **GenericConduit for extended databases**. The wizard will generate project files and source code based on the C++ Generic Conduit Framework for **extended database**s.
7. Under **Desktop file format**, click **XML**. Your conduit will synchronize with an XML file that is structured to represent any extended database. Use any text or XML editor to easily make and view modifications to this file.
8. Under **Registration**, click **Folder registered**. Your conduit will be registered with HotSync Manager simply by placing your conduit DLL in the `Conduits` folder at install time. For folder-registered conduits, you must also specify the following:
   - **Creator ID**: the key by which HotSync Manager identifies conduits. You must specify a value. Use the Conduit Wizard's default value of `'memO'` in this quick-start guide (the last character is the capital letter "O").
   - **Remote DB**: the name of the database on the handheld that this conduit synchronizes. Specifying a value is optional.

   These are the values that your conduit passes back to HotSync Manager via the *`pOut`* parameter when it calls your conduit's `GetConduitInfo()` entry point and passes in *`infoType`* = `eRegistrationInfo` at run time.

NOTE: When you choose **Folder registered**, the Conduit Wizard sets the output path and filename of your conduit to `C:\Documents and Settings\All Users\Application Data\HotSync\Conduits\$(ProjectName).dll`. So after you build your conduit, it will automatically be registered for the system (all Windows users) and ready to be run without further action on your part.

---

9. Under **Details**, select the check box labelled **Do not require an application on the device with this conduit's creator ID for this conduit to run**. In this quick-start guide, you will have a matching application on the handheld, so it doesn't really matter what this option is set to. However, for the conduit you ship, carefully consider whether your conduit always runs or not; if no application is present, then your

conduit might be creating orphaned databases that the user doesn't need.

With this setting, your conduit passes back 1 when HotSync Manager calls your conduit's `GetConduitInfo()` entry point and passes in *`infoType`* = `eRunAlways`.

10. Under **Details**, clear the check box labelled **Do not display this conduit's name in the HotSync Progress dialog box on the desktop**. Seeing your conduit's name in this dialog box will help confirm whether your conduit is being run.

    With this setting, your conduit passes back 0 when HotSync Manager calls your conduit's `GetConduitInfo()` entry point and passes in *`infoType`* = `eDoNotDisplayInConduitListForUser`.

11. Review your choices and click **Finish**.

At this point, the Conduit Wizard has created a project file, solution file, header files, and source files. The `MyConduit.vcproj` project file is open in VC and ready for you to proceed to the next step.

Step 2: Prepare Palm OS Cobalt Simulator

This step describes how to install the CDK's Memo Port application on the Palm OS Cobalt Simulator. This is a version of the classic Memo Pad application that uses an extended database rather than a classic database. The conduit you build in the next step will synchronize with the extended database created by this application.

Configure Simulator for a Network HotSync Operation

Later in this quick-start guide, you will perform a network HotSync operation with Palm OS Cobalt Simulator to see your conduit operate. But first you must install and configure Simulator and you must configure HotSync Manager. To do this, follow the steps outlined in "Configuring Simulator for Network Connection" on page 55 in *Conduit Development Utilities Guide*.

Disable Time-Outs

Simulator (and a real handheld) times out if a HotSync operation does not complete within a short period of time—which can interrupt your debug session. To disable time-outs, see "Disabling Time-outs" on page 61 in the *Conduit Development Utilities Guide*.

Install Memo Port on Simulator

The CDK includes the `MemoPort.prc` file to install on a Palm OS Cobalt handheld or Simulator, as well as the `MemoPort.dll` file required to run this application on Palm OS Cobalt Simulator. Follow these steps to install the application on Simulator:

1. Locate the `MemoPort.prc` and `MemoPort.dll` files in `<CDK>\C++\Win\Samples\GenericConduit\SamplePrc\PalmSim`.
2. Copy these files to the same directory as the Palm OS Cobalt Simulator executable: `PalmSim.exe`.
3. Start Palm OS Cobalt Simulator.
4. In Simulator, right-click to display the context menu and select **Install** > **Database**.
5. Browse to the Simulator folder, select the `MemoPort.prc` file, and click **Open**.

Create Sample Data with Memo Port

Create some memos in the Memo Port application so that, after you perform a HotSync operation, you will be able to see these memos on the desktop.

6. In Palm OS Cobalt Simulator, start Memo Port.
7. Click **New** and type a short memo. Create additional memos if you wish.

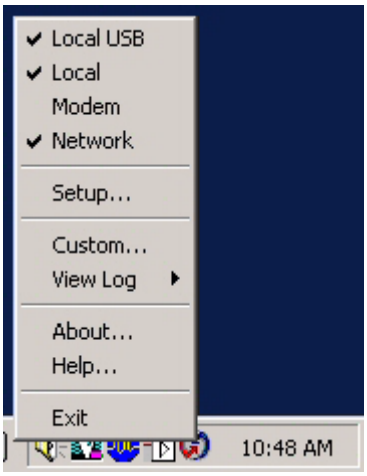
Step 3: Configure the Project Settings and Build the DLL

In this step, you will make minor changes to your VC project's settings and build your conduit DLL.

1. If the HotSync Manager icon is present in the Windows taskbar (in the lower right of your screen), click the HotSync Manager icon and click **Exit** as shown in Figure A.3.

   The project created by Conduit Wizard will automatically launch the HotSync Manager in the CDK folder when you debug your conduit, so exit HotSync Manager first, if it is already running.

Figure A.3 Exit HotSync Manager



2. In the VC **Solution Explorer**, select the **MyConduit** project (not the "Solution") and click **Project > Properties**.
3. In the **MyConduit Property Pages** dialog box under **Configuration Properties**, click **Debugging**.
4. In the **Command Arguments** box, enter the following HotSync Manager command-line argument as shown in Figure A.4:

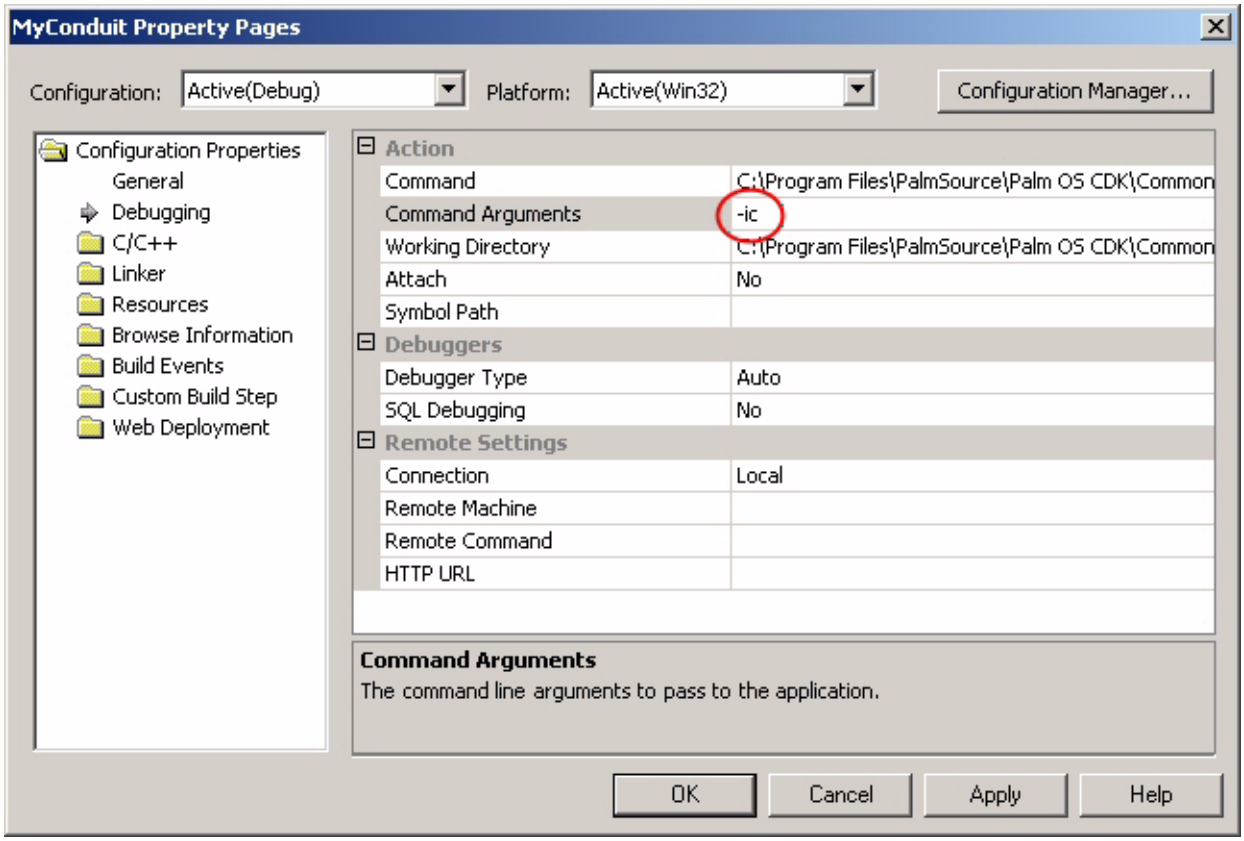
   ```
   -ic
   ```

   This argument causes HotSync Manager to launch the Conduit Inspector, a developer utility that logs detailed

HotSync Manager status in real time. It is not a requirement that you run it, but it can help you debug problems with your conduit.

5. Click **OK**.

Figure A.4 Changes to Property Pages > Debugging



6. Click **Build > Rebuild Solution** to build your conduit.

At this point, your conduit (`MyConduit.dll`) is built. And it is already registered and ready to be run, because it is located in the system's `Conduits` folder (`C:\Documents and Settings\All Users\Application Data\HotSync\Conduits\MyConduit.dll`) and because its `GetConduitInfo()` entry point responds correctly when HotSync Manager passes in *`infoType`* = `eRegistrationInfo` at run time.

Now you're ready to debug your conduit.

Step 4: Single-step/Debug Your Conduit

This step provides a starting point for you to single-step through the conduit code and debug it. The sections are:

- Start Debugging
- Check the Custom Dialog Box
- Perform a HotSync Operation
- Save Conduit Inspector Logs

Start Debugging

1. In VC, open the `entrypoints.cpp` file and set breakpoints at the following conduit entry functions:
   - `OpenConduit()`
   - `GetConduitInfo()`
   - `CfgConduit()`
2. To start debugging, select **Debug** > **Start** (or use the keyboard shortcut: press F5). You might see the following warning:

   'Hotsync.exe' does not contain debugging information. (No symbols loaded.) Click OK to debug anyway.

   This is normal, so click **OK** to continue.
3. When the Conduit Inspector window appears, size it and place it so you can see it beside VC. Conduit Inspector logs in real time many useful messages about what HotSync Manager and your conduit are doing. Figure A.5 shows Conduit Inspector's opening window.

   If Conduit Inspector does not start, refer to Figure A.4 on page 179 and confirm that you set the program arguments as shown.
4. Conduit Inspector displays a **HotSync Realtime Log** in Figure A.5.

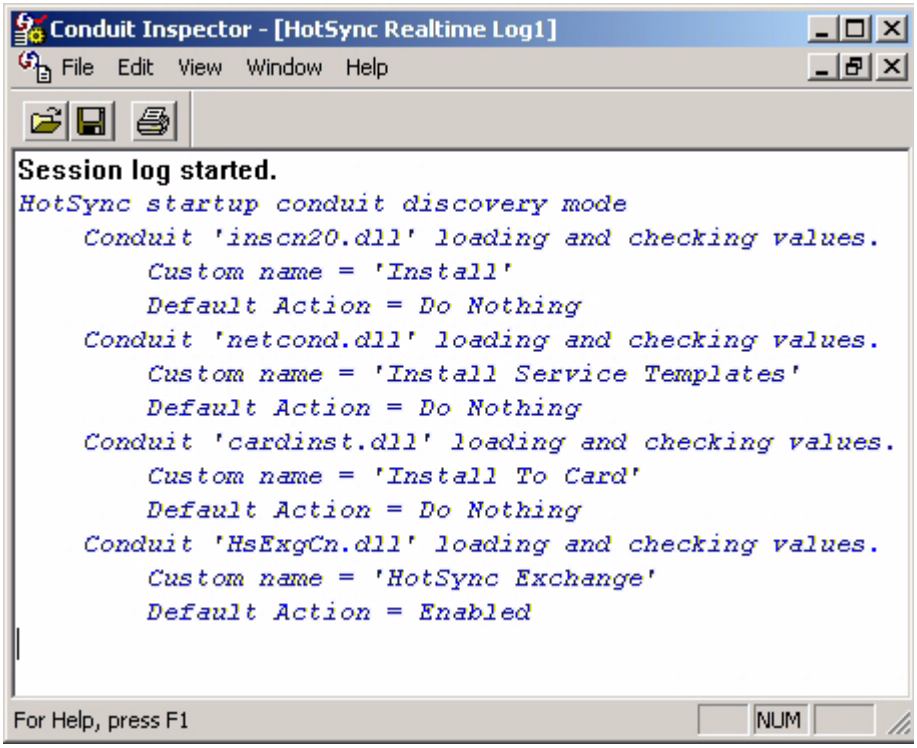
   See Chapter 6, "Conduit Inspector Utility," on page 29 in the *Conduit Development Utilities Guide* for details on using Conduit Inspector.

5. At the same time Conduit Inspector starts, VC reaches the `GetConduitInfo()` breakpoint. Press F10 to single-step or press F5 to continue. For each conduit, HotSync Manager calls this entry point several times, so you must press F5 once for each time call before HotSync Manager completes its startup.

   If you see alerts from HotSync Manager that say "The Notifier file 'filename.dll' was not found," you can safely click **OK** and ignore them. To avoid seeing this alert, you can use the CondCfg utility to unregister them or execute hotsync.exe -r to remove the registration of any nonexistent notifiers.

NOTE: If you have used this quick-start guide before, you will see the **Choose Conduit** dialog box. HotSync Manager displays this dialog whenever more than one conduit has the same creator ID. Select the conduit you want to run and click **OK**. If you click **Ignore**, HotSync Manager will not run your conduit. The conduits that you do not choose are moved to the `Conduits\Disabled` folder.

Figure A.5 Conduit Inspector describes what conduits and HotSync Manager do in real time



Check the Custom Dialog Box

The following steps are a quick test to ensure that HotSync Manager is calling your conduit and executing its `CfgConduit()` entry point. The `CfgConduit()` entry point is called when the user clicks **Custom** from the HotSync Manager menu, selects your conduit, and clicks **Change**. The following steps show how to check your Custom dialog box:

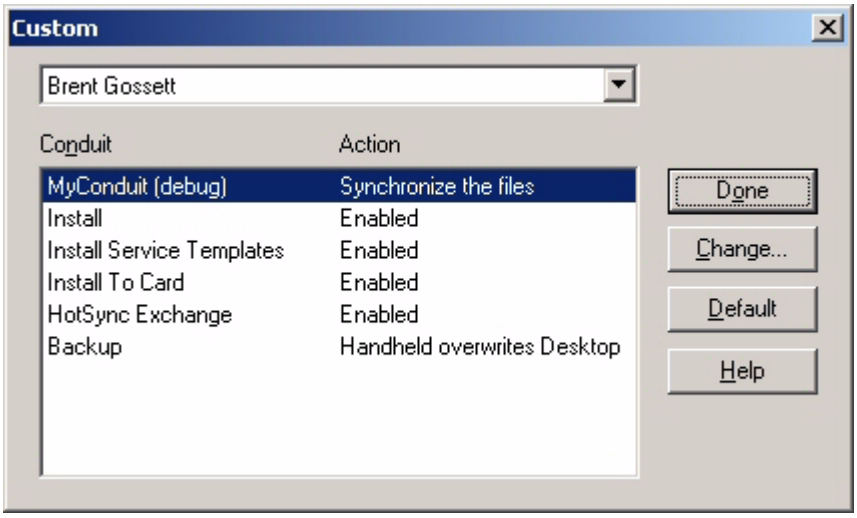
6. Click the HotSync Manager icon (as shown in Figure A.3), and click **Custom**.

   Before the **Custom** dialog box appears, note that VC reaches your conduit's `GetConduitInfo()` entry point. HotSync Manager queries your folder-registered conduit for information to display in the **Custom** dialog box.

7. Press F5 once for each call to `GetConduitInfo()` that HotSync Manager makes.

   After HotSync Manager calls this entry point several times, it displays the **Custom** dialog box as shown in Figure A.6.

Figure A.6 HotSync Manager Custom dialog box

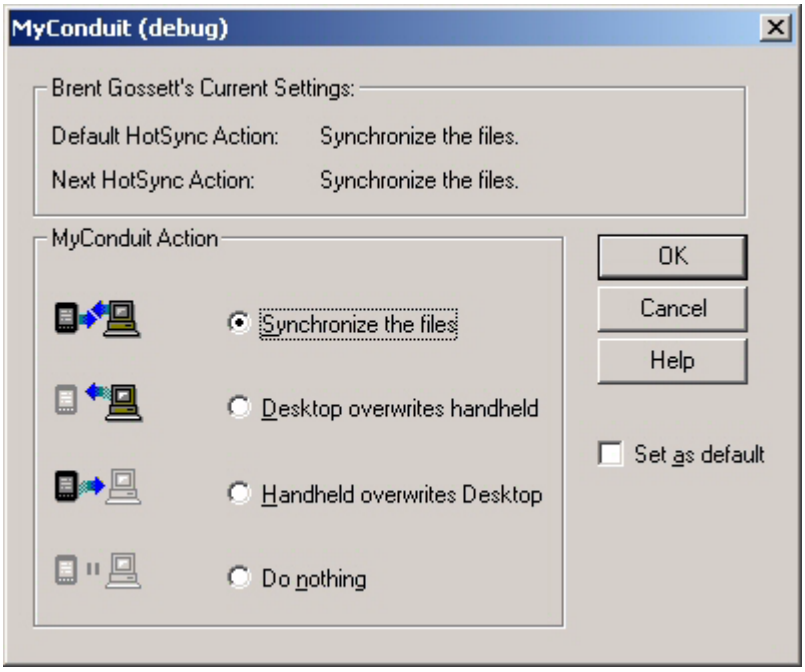


8. In the **Custom** dialog box, select **MyConduit**, and click **Change**.

   Ensure that in VC you reach the `CfgConduit()` breakpoint.
9. Press F5 to continue. The **MyConduit (debug)** dialog box appears as shown in Figure A.7. This dialog box allows the user to change temporary or permanent synchronization preferences for this conduit.
10. Dismiss all HotSync Manager dialog boxes, **MyConduit (debug)** and **Custom**.

This confirms that HotSync Manager is calling your conduit and executing its `CfgConduit()` entry point.

Figure A.7 Conduit's synchronization preferences dialog box



Perform a HotSync Operation

11. Start the Palm OS Cobalt Simulator, if you have not done so already.
12. In Simulator, start the **HotSync** client application.
13. In the HotSync client, click the **Network** button.

14. In the HotSync client, click the **HotSync** button.

    In VC, check that you reach the `GetConduitInfo()` breakpoint.

    Notice that Conduit Inspector opens an additional real-time log window and logs the details of this HotSync operation.

    The HotSync Progress dialog box appears briefly.

NOTE: If HotSync Manager does not appear to respond, then see "Configure Simulator for a Network HotSync Operation" on page 176 and confirm that you have set up Simulator and HotSync Manager correctly.

---

15. In VC, press F5 to continue. HotSync Manager calls your conduit's `GetConduitInfo()` entry point several times. Press F5 once each time.

    If you have not previously performed a HotSync operation with this Simulator session, then the **Create or Select HotSync User** dialog box appears. If you have, then skip to step 17.

16. Create a HotSync user account or select an existing one and click **OK**.

    The HotSync operation continues. In VC note that you reach the `GetConduitInfo()` entry point again.

17. Press F5 to continue from `GetConduitInfo()`.

    In VC, you reach the `OpenConduit()` entry point.

18. Press F5 to continue from `OpenConduit()`.

    The HotSync operation finishes. Conduit Inspector has logged what HotSync Manager and all of the conduits did during the HotSync operation.

19. At this point, you can continue or stop debugging. To stop debugging in VC, click **Debug** > **Stop Debugging**.

    Notice that Conduit Inspector remains open for you to use, but HotSync Manager exits.

NOTE: After finishing a debug session, HotSync Manager does indeed exit. However, its icon might remain visible in the Windows taskbar until you pass over it with your pointer.

Save Conduit Inspector Logs

At any time, save your Conduit Inspector log files to review later.

20. Select an open **HotSync Realtime Log** window and select **File** > **Save As** to save the log file.
21. If any other log windows are visible, save them as well.

When you are finished debugging, you can proceed to the next step to restore your conduit settings to what they were when you started.

Step 5: Examine the MyConduitData.xml File

When you performed a HotSync operation in the previous step, your conduit synchronized the MemoDB database on Simulator with an XML file on the desktop located here:

```
C:\Documents and Settings\
<Windows User>\My Documents\Palm OS Desktop\
<HotSync User>\MyConduit\MyConduitData.xml
```

Open this file with your favorite XML editor and note that the memos you created in Simulator ("Create Sample Data with Memo Port" on page 177) is present.

To experiment further, edit the `MyConduitData.xml` file, perform a HotSync operation, and note the change in the Memo Port application in Simulator.

NOTE: When you make changes to data in the XML file, be sure to set the "Modified" flag. Otherwise, the sync logic in the Generic Conduit Framework will not recognize that a change has been made and will not transfer the change to the database.

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W