Welcome to

Developing Palm OS Conduits

08.96

Navigate this online document as follows:

To see bookmarks	**Type Command-7**
To see information on Adobe Acrobat Reader	**Type Command-?**
To navigate	**Click on** **any blue hypertext link** **any Table of Contents entry** **arrows in the menu bar**

©1996 U.S. Robotics, Inc. All rights reserved.

Documentation stored on the compact disk may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from U.S. Robotics.

U.S. Robotics, the U.S. Robotics logo and Graffiti are registered trademarks, and Palm Computing, HotSync, Palm OS, and the Palm OS logo are trademarks of U.S. Robotics and its subsidiaries.

All other trademarks or registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK ARE SUBJECT TO THE LICENSE AGREEMENT.

U.S. Robotics, Palm Computing Division
Mail Order
1-800-881-7256

U.S. Robotics, Palm Computing Division
World Wide Web site: `http://www.usr.com/palm`
Registration information (Internet): `register@metrowerks.com`
Technical support (Internet): `devsupp@palm.com`

Table of Contents

Table of Contents iii

1 Getting Started 9

What's a Conduit? 9
What Are Development System Requirements? 10
What's in the Conduit SDK? 11
Overview of the Conduit SDK 11
Top-Level Directories 11
SDK Development Directories 12
Directories 12
Files 12
Conduits Sample Source Code Directory Contents 12
What About HotSync1.1 13
What's in This Guide? 13

2 Conduit Basics 15

Basic Approaches to Conduit Design 15
Conduit Basic Control Flow 16
Locating Records on the Device. 19
Minimum Conduit Requirements 21
Registering the Conduit 21
Providing C Entry Points 21
Providing a DllMain() Routine 21
Sending Errors and Other Messages 22
SyncManager Memory Management 24
Structures with Dynamically Allocated Memory: 24
Conduits and the Windows Registry 24
Naming Third Party Conduits 25
Registering Third Party Conduits. 25
Providing the Conduit Name 25
Providing Name/Data Pairs. 26
Registry Entry Example. 28
Default Registry Keys. 28
Installing and Removing Your Conduit 29

Installing Your Conduit 29
HotSync 1.1 Installation. 29
Conduit Installation 31
Removing Your Conduit 31
Cable vs. Modem Connection 31
FastSync and SlowSync 32

3 Conduit Design Decisions 35
Conduit Design Questions 36
Using the Native Synchronization Logic 38
Pilot Desktop OS Native Synchronization Algorithm 38
Record-Level Synchronization with Pilot Applications . . . 40
Archiving Records 41

4 Control Flow of Pilot Desktop's Native Synchronization Logic . 43
Basic Control Flow 43
Functions Called During Synchronization 44
Synchronizing with Existing PC Applications. 45
Synchronizing Categories 46

5 Implementing a Conduit 47
Providing “C” Entry Points 47
Providing a DllMain Routine 48
Providing Entry Point Routines 50
The OpenConduit Function 50
The GetConduitName Function 52
The GetConduitVersion Function 52
Creating a CBaseMonitor Subclass 53
CBaseMonitor Basic Structure 53
CBaseMonitor Data Members 55
CBaseDTLinkConverter* m_pDTConvert 55
PROGRESSFN m_pfnProgress. 55
CBaseTable* m_LocRealTable 56
CBaseTable* m_LocArchTable 56
CBaseTable* m_BackupTable 56
CBaseTable* m_RemRealTable 56
CSyncProperties m_rSyncProperties 57

CCategoryMgr* m_LocCategory 57
CCategoryMgr* m_RemCategory 57
BYTE m_RemHandle 57
char m_ArchFileExt[5] 57
int m_TotRemoteDBs 57
int m_CurrRemoteDB 58
CDbGenInfo m_DbGenInfo 58
HINSTANCE m_DllInstance 58
CBaseMonitor Functions Must to Override 58
Monitor Constructor and Destructor 60
ObtainLocalTables 61
ObtainRemoteTables 62
AddRecord 64
AddRemoteRecord 65
ChangeRemoteRecord 66
CreateLocalArchTable 68
FastSyncRecords 69
SlowSyncRecords 71
CopyRecordsPCtoHH 74
CopyRecordsHHtoPC 76
LogRecordData 78
LogApplicationName 79
CBaseMonitor Functions You May to Override 79
SaveLocalTables 80
PurgeLocalDeletedRecs 81
ApplyRemotePositionMap 83
Creating a CBaseDTLinkConverter Subclass 84
CBaseDTLinkConverter Basic Structure 84
The Log Object 85
Casting of Member Functions 85
Carriage Returns and Line Feeds 85
CBaseDTLinkConverter Data Members 85
CSyncLog* m_pLog 86
TCHAR* m_TransBuff 86
HINSTANCE m_DllInstance 86
CBaseDTLinkConverter Functions You Must Override . . . 87

CAddressDTLinkConverter Constructor and Destructor . 87
ConvertToRemote 88
ConvertFromRemote 90
ConvertToRemoteCategories 92
ConvertFromRemoteCategories 94
CBaseDTLinkConverter Functions You May Override. . . . 95
CBaseDTLinkConverter Utility Member Functions 95
Creating a CBaseTable Subclass. 97
How to Set Up Tables 97
More About Tables 99
CBaseTable Class 99
CBaseRecord Class. 102
CBaseSchema Class 104
CBaseIterator Class 104
Considering Category Manager Modifications 105

6 SyncManager Function Calls. 109
Session-Oriented Calls 109
SyncRegisterConduit 109
SyncUnRegisterConduit. 110
File-Oriented Calls 110
SyncCloseDB 112
SyncCreateDB 112
SyncDeleteDB 113
SyncOpenDB 114
SyncReadDBAppInfoBlock 115
SyncReadDBSortInfoBlock. 116
SyncResetSyncFlags 117
SyncWriteDBAppInfoBlock 118
SyncWriteDBSortInfoBlock 118
Record-Oriented Calls. 119
SyncDeleteAllResourceRec 120
SyncDeleteRecord 121
SyncDeleteResourceRec 121
SyncGetDBRecordCount 122
SyncPurgeAllRecs 122

SyncPurgeDeletedRecs 123
SyncReadNextModifiedRec 123
SyncReadRecordById 124
SyncReadRecordByIndex 125
SyncReadResRecordByIndex. 125
SyncWriteRec 126
SyncWriteResourceRec 127
Utility Calls 128
SyncReadDBList 128
SyncReadSingleCardInfo 129
SyncReadSystemInfo 130

7 Error Codes 133
SyncManager Return Codes 133
SyncManager Fatal Return Codes 134
SyncManager Base Class Return Codes 135

1

Getting Started

This chapter helps you get started with conduit design by providing an overview of the available software and a roadmap to the conduit design process described in this manual. This chapter answers the following questions:

- What's a Conduit?
- What Are Development System Requirements?
- What's in the Conduit SDK?
- What About HotSync1.1
- What's in This Guide?

What's a Conduit?

A conduit is a dynamic link library (DLL) running under Microsoft Windows. Conduits exchange and synchronize data between an application running on a PC under Windows and an application running on the Pilot organizer or another Palm OS based device.

End-users can push the HotSync button on the cradle to request synchronization of data between all device applications and the corresponding Windows applications. To do so, the end-user must perform the following two steps:

1. Insert the device into its cradle, which has to be connected to the PC with a serial cable
2. Press the HotSync button on the cradle

The HotSync program, which runs under Windows, synchronizes each application by executing its conduit.

Many conduits (including the conduits for the four native applications on the first device) synchronize data between the device and the PC to be mirror images after synchronization. Other conduits perform more complex operations. The complexity of your conduit's behavior determines the development effort involved.

To make things easier for you, part of the conduit SDK consists of several C++ classes that provide predetermined functionality that you may be able to customize to suit your needs. The four applications included on the first Pilot device (Date book, address book, ToDo list, and MemoPad) use conduits based on those classes and the associated synchronization logic. Source code for each of the four native applications is part of the SDK.

If your application doesn't sync with one of the four native Pilot applications, or your application's behavior is so different from the existing conduits behavior (the native synchronization logic) that customizing becomes impractical, you can still take advantage of the SyncManager API. In that case, you should do the following:

- Read, at a minimum, the chapter Conduit Basics in this document. You may find it helpful to look at other chapters as well.
- Look at the documentation in SyncManager Function Calls
- Examine a small example conduit (\poscond\txtcond) with more simple behavior to write your conduit from scratch

Conduits are AFX extension modules; see MFC Tech Note 33 for more information. Note that the sample Makefile provided with the Conduit SDK automatically makes your conduit an AFX extension.

What Are Development System Requirements?

Conduits are developed using standard Microsoft Visual C++ tools. To install and use the Conduit SDK for Windows, your system must be equipped with:

- Windows 95 or Windows NT
- Visual C++ version 4.0 or greater
- Microsoft MFC 3.0 or greater
- At least 16 MB RAM and 5 MB free disk space
- Pilot with cradle for testing
- Adobe Acrobat Reader for viewing the online documentation (included in the SDK)

What's in the Conduit SDK?

This sections starts with an Overview of the Conduit SDK, then lists relevant parts of the Top-Level Directories and the SDK Development Directories

Overview of the Conduit SDK

The conduit SDK provides the following header files, libraries, and sample code that you need to develop a conduit for your Palm OS application:

- **Header Files**—C and C++ header files with structure definitions and function prototypes
- **API Libraries—**A set of libraries that provide access to device data.
- **HotSync 1.1 and HotSync 1.1 Libraries**—The executable loads and executes conduits, the libraries must be present at runtime for conduits to execute properly. Both debug and release versions are included.
- **Sample Code**
  - Source code for a conduit that synchronizes memo pad data by storing it as a simple ASCII text file on the PC
  - Source code for the four conduits included with the first release of the device (address book, memo pad, to do list, date book)

This section provides an overview of important folders and files in the SDK. See the Installation Instruction for information on how to install the SDK.

Top-Level Directories

During installation, the SDK creates the following directory structure on your system:

\POSCOND\—Main directory

CONDSDK—SDK development files and documentation
TODCOND—Code for the sample ToDo conduit
TXTCOND—Code for the sample text file conduit
ADDCOND—Code for the sample Address Book conduit
DATCOND—Code for the sample Date Book conduit
MEMCOND—Code for the sample Memo Pad conduit

SDK Development Directories

Directories

\POSCOND\CONDSDK

\INCLUDE—Header files for development
\SRC—Source files used for development
\HELP—Help file for the HotSync program
\DOCS—Conduit SDK documentation in .pdf format
\BIN—Debug and release build of libraries and HotSync1.1

Files

\POSCOND\CONDSDK\INCLUDE

\abrecord.h—Address Book record class definition \dataprv.h—Definitions and structs for Pilot database records \syncmgr.h—SyncManager public API and structures
\updcatid.h—Object used for updating categories
\logstrng.h—#defines for resource string IDs
\catmgr.h—Category manager class definitions
\basemon.h—BaseMonitor class definition
\synclog.h—CSyncLog class definition
*Native*table.h—Table subclass definition for native app
*Native*record.h—Record class definition for native app
\basetabl.h—Base table class definition
\bfields.h—Field objects class definitions
\basemon.rc—Resource file containing error and log strings
\basemon.cpp—Source code to the base monitor class

Conduits Sample Source Code Directory Contents

The following is a list of files for the ToDo conduit; the conduits of the other native applications have the same structure.

\POSCOND\TODCOND

\todlink.h—ToDo link converter class definition
\todlink.cpp—ToDo link converter source code
\todmon.h—ToDo link monitor class definition
\todmon.cpp—ToDo link monitor source code
\todcond.h—ToDo DLL header file
\todcond.cpp—DLLMain() and 'C' entry points source code
\todcond.mak—ToDo conduit make file

The text conduit is included as a simple example for developers who decide not to use the native logic.

\TXTCOND
\STEP01—A simple text file transfer to the Pilot MemoPad
\STEP02—More advanced version of Step01 source code

What About HotSync1.1

When you ship your conduit to end-users, you need to include HotSync1.1, the 1.1 conduits for the four native Pilot applications, and the appropriate installation procedure on the installation disks. You must do this because the HotSync Application included with the Pilot Desktop 1.0 was compiled with VC++ 2.2 and MFC 2.0. Because you are most likely using VC++ and MFC 4.0, your application won't run with HotSync 1.0. For more information, see HotSync 1.1 Installation.

What's in This Guide?

This guide serves as a programming guide and a reference manual. It helps you make design decisions and provides structure and function descriptions to help you implement your design.

This manual contains the following:

- Conduit Basics—Provides a top-level overview of the behavior of any application. It explains how HotSync interacts with the different conduits on the system to synchronize each.
- Conduit Design Decisions—Points to some critical design decisions you have to make early. This includes a list of questions that help you determine whether using the available C++ class hierarchy makes sense.
- Control Flow of Pilot Desktop's Native Synchronization Logic—Describes how the four built-in applications use the C++ classes to implement their synchronization behavior
- Implementing a Conduit—Steps you through implementing a conduit based on the available C++ classes. This chapter includes descriptions of the classes you have to subclass and their data members and member functions.
- SyncManager Function Calls—Provides complete description of the API calls that all conduit applications can use
- Error Codes—Lists the error codes that can be triggered when calling SyncManager functions

2 Conduit Basics

This chapter describes what every conduit must do to successfully synchronize a device application with the PC. This chapter provides an overview of the control flow inside the simplest possible conduit library. When you design a conduit, it's essential that you understand how a synchronization process works and which part of the system is responsible for corresponding synchronization process.

To provide information that is applicable no matter how complex your conduit is, this chapter usually assumes a simple conduit that links an application on the Palm OS device to a simple text file on the PC. While this kind of synchronization is not realistic for most applications, it is helpful for getting started; a sample conduit library that performs exactly that kind of synchronization is included in your software development kit.

The sample conduit is called txtcond. Two versions are included, one more complex than the other. They both import and export memo pad data to and from text files on the PC.

This chapter discusses the following topics:

- Basic Approaches to Conduit Design
- Conduit Basic Control Flow
- Minimum Conduit Requirements
- SyncManager Memory Management
- Conduits and the Windows Registry
- Installing and Removing Your Conduit
- Cable vs. Modem Connection
- FastSync and SlowSync

Basic Approaches to Conduit Design

Conduit developers generally take one of the following approaches to conduit design:

- If the conduit is relatively similar to the four conduits included with the Pilot Desktop, you can use the existing C++ classes and modify the source code of one of the conduits appropriately.
  - Conduit Design Decisions helps you decide whether using the native synchronization logic makes sense for you.
  - Chapter 5, Implementing a Conduit explains what you need to do to implement such a conduit.
- If the conduit requires behavior that the native conduits don't deal with, it might make more sense to develop the conduit from scratch. In that case, the simple sample conduit (textcond) provides a useful starting point. You can use the API documented in SyncManager Function Calls to implement your conduit.

Conduit Basic Control Flow

When the user presses the HotSync button on the device, the following events occur:

1. HotSync looks for the User ID transmitted by the Palm OS device and compares it with the local Pilot user database. It finds the user ID on the device or creates a new user ID.
2. Upon startup, HotSync looks at the list of conduits in the registry and prepares a list of their creator IDs.

   Each conduit has to enter appropriate registry information when the application is first installed see Installing and Removing Your Conduit.
3. HotSync determines whether to do a FastSync or a SlowSync based on whether the device was previously synchronized with the same PC or a different one. If the device was last synchronized with a different PC, the modification flags on the device are not accurate with respect to the current PC.

   The native synchronization logic in the four native conduits takes advantage of this distinction by using an optimized FastSync algorithm whenever it can. This algorithm only considers records on either side that have one of the modification flags set (dirty or deleted). If the flags are not reliable, a SlowSync algorithm is used that examines each record on both sides of the sync.
4. HotSync queries the Palm OS device for all databases that do not have one of the executable system types, such as `sysFileTApplication`. These databases are matched up with the creatorIDs from the registry.

5. HotSync starts the synchronization process.
6. For each creatorID that's found in the registry, HotSync passes the `SyncProperties` class to the matching conduit including the name of the first matching database found on the Palm OS device. At this point, HotSync passes control to the conduit until the conduit returns when synchronization of that application is complete or had to be aborted.

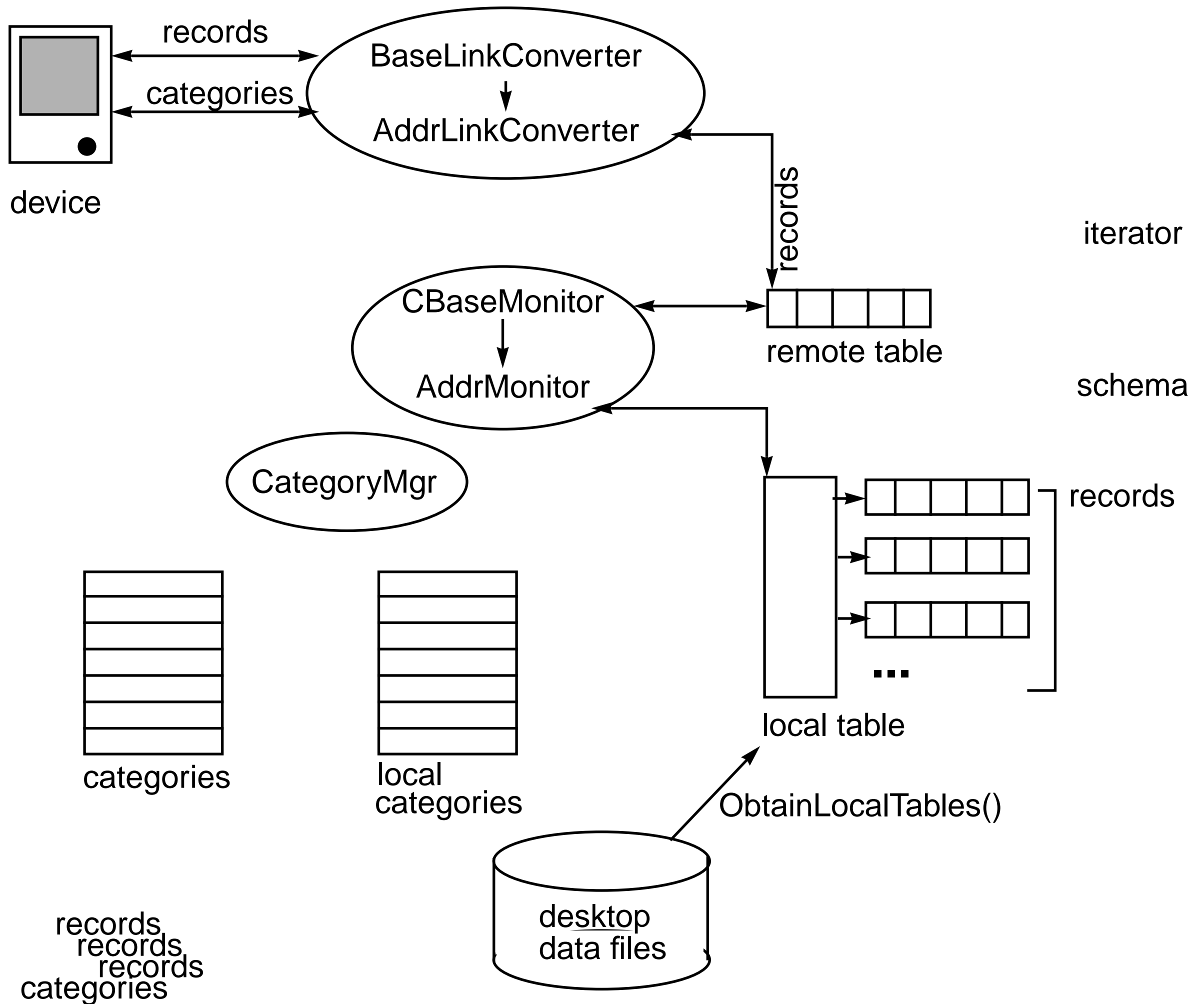
   It is the conduit's responsibility to retrieve any required databases—other than the one passes with the SyncProperties object—from the Palm OS device.
7. After HotSync has iterated through all conduits in the registry, it calls the backup conduit and install conduit.
   - If there is a database on the device for which the backup flag is set but no conduit exists, the backup conduit provided by the system copies the data from the device into the BACKUP directory in the user's area on the PC. The file format is the same as on the device. Note that this backup conduit does not check whether data have been modified and will therefore execute each time.

     While setting the backup bit may be appropriate for small data databases, it's not recommended for applications or large data databases. For large data databases, create a specific conduit.
   - The install conduit works together with the AppInstaller provided by Pilot Desktop to install applications or other databases onto the Palm OS device. The App Installer places a copy of the database to be installed in the user's Install directory and puts some installation information in the user database and the registry (or /.ini) file. The install conduit uses this information to copy the database down to Pilot, replacing any existing copies.

     In the event of a hard reset, the install conduit is used to restore databases that have been previously been backed up and are residing in the user's Backup directory.
8. The conduit performs synchronization using calls from the SyncManager library (see SyncManager Function Calls).

   As the conduit performs the synchronization, it must pay attention to the following:
   - Minimum Conduit Requirements. Every conduit must publish three "C" entry points to be invoked by HotSync (see Providing C Entry Points and Providing a DllMain() Routine.

- Sending Errors and Other Messages. A conduit should log errors and other information using the log object. This helps HotSync determine whether synchronization was successful and whether the log should be displayed to the user.
- Appropriate updates. Each conduit has to decide on proper updates of records on both the device and the PC depending on their current status. Using the Native Synchronization Logic explains how the four applications included on the Pilot device perform synchronization. Your application may use a different logic based on the information each record contains.



Locating Records on the Device

Database records on the device consist of the following two distinct parts:

- The first part is a **fixed-length** portion containing the record ID, a status field (indicating Add, Modify, Delete, or Archive status for the native applications), and a category ID field
- The second part is of **variable length** depending on the number of fields and whether they contain any data

Because the SyncManager DLL acts as a channel for byte traffic to and from the device, a generic structure that handles any record format is needed. This generic structure then becomes a parameter in the record-oriented API.

To locate remote records, three different APIs are provided, allowing you to do the following:

- Sequentially locate the next altered record using SyncReadNextModifiedRec.
- An exact record lookup using SyncReadRecordById.
- Top to bottom iteration using SyncReadRecordByIndex.

The same object, `CRawRecordInfo`, is used by all three functions. However, different members of the object are used by each function call to help indicate the nature of the remote lookup activity.

Figure 2.1 illustrates where the fixed-length data from a device record is stored in the `CRawRecordInfo` object and also shows that the data member `m_pBytes` points to the variable-length record body. These `CRawRecordInfo` structure members are populated by the record-oriented API when a record has been retrieved successfully.

```
class CRawRecordInfo{
public
        BYTE        m_FileHandle;  ┐
        DWORD       m_RecId;       │
        WORD        m_RecIndex;    │   fixed length
        BYTE        m_Attribs;     │
        short       m_CatId;       │
        int         m_ConduitId;   │
        DWORD       m_RecSize;     │
        WORD        m_TotalBytes;  ┘
        BYTE*       m_pBytes; ────────► variable length
```

Figure 2.1 CRawRecordInfo Structure Pointing to Record Information

Minimum Conduit Requirements

If the predefined C++ classes and the associated synchronization logic is not appropriate for your application, the only requirements (recommended) to have HotSync accept your conduit include the following:

- Registering the Conduit
- Providing C Entry Points
- Providing a DllMain() Routine
- Sending Errors and Other Messages (strongly recommended)

Registering the Conduit

You must insure that information about the conduit is included in the windows registry when users first install your application.

See Conduits and the Windows Registry for details on the information you must enter into the Registry and Installing and Removing Your Conduit for information on how to provide it.

Providing C Entry Points

Every conduit must publish the following three "C" entry points to be invoked by HotSync:

- `OpenConduit`
- `GetConduitName`
- `GetConduitVersion`

All native conduits carry out all synchronization duties within `OpenConduit` before returning control to HotSync. HotSync invokes `OpenConduit` only once, immediately after it has dynamically loaded the conduit into memory. See Providing "C" Entry Points for more detailed information.

Providing a DllMain() Routine

Your application must provide a typical 32-bit Windows `DllMain()` routine. The Windows operating system automatically calls `DllMain()` when it loads the conduit DLL into memory because HotSync performs a `LoadLibrary()` call on it.

Note that the Makefile provided for compiling the library makes it an AFX extension library, which means that certain classes outside the included files are available to your application.

Sending Errors and Other Messages

When HotSync starts a conduit, it passes a pointer to a log object to it. The object is a member of the `SyncProperties` class named `m_pSyncLog` and is used to store error messages and other information for the end-user.

Note that a C-based conduit can use the log object like it would use a structure, but it is still an object and the library therefore has to be compiled as C++.

The `CSyncLog` object has the following public interface:

Listing 2.1 CSyncLog class

```
class CSyncLog {
  public:
    CSyncLog(int nFlushThreshold = 0);
    ~CSyncLog();
    LogError AddEntry(   const char* pszEntry,
                         Activity act=slText,
                         BOOL bTimeStamp = FALSE);
      LogError SaveLog(const char* pszLogFile);
    BOOL BuildRemoteLog(CString& csRemoteLog);
    void GetWorkFileName(CString& csWorkFileName);
    void CloseLog();
    WORD TestCounters(); };
```

When all conduits have completed, HotSync saves the log object to disk.

The member function most often used by a conduit to log information is the `AddEntry()` routine. In its simplest form, a string may be recorded into the log:

```
AddEntry("Simple Line of Text").
```

The other parameters to `AddEntry` have default values that the caller may but doesn't have to override:

- The `act` parameter is a member of the Activity enum defined in SYNCLOG.H and discussed below.
- The `bTimeStamp` parameter signals the log object to time stamp the new entry as it is added to the log. In most cases, you can leave this parameter undefined (the default).

Here's some information about the values you're most likely to supply as the `act` parameter, and how to use them:

- slSyncStarted—send at the beginning of the synchronization process, don't supply a text string. This is required so the log knows you are logging a new conduit.
- slSyncFinished—send at the end of the synchronization process and pass in the name of the application.
- slSyncAborted—requests that HotSync put up banner that a problem occurred.
- Other enum values—these will signal HotSync to display banner to the user at the conclusion of the sync. They may be passed with a text string to be included in the log file.

The following example code shows a conduit reporting that it encountered a problem adding a new record to the remote device database.

Listing 2.2 Error Logging Example

```
if (AddRemoteRecord(rLocRecord)!= 0)
    {
    char errBuff[MAX_LOG_STRING];
    strcpy(errBuff, "Could not Add the Smith
                    address record");
    m_rSyncProperties.m_pSyncLog->AddEntry(
        (const char*)errBuff, slRemoteAddFailed);
    }
```

SyncManager Memory Management

The SyncManager carries out commands and returns replies from the device. When supplying or retrieving data accompanying to these commands, the SyncManager acts as a channel for this raw byte traffic.

Some of the objects that are passed as parameters between functions in the SyncManager and a Conduit.DLL contain dynamically allocated memory. These objects contain a generic data area (usually named `m_pBytes`) where the SyncManager places raw data obtained from the device. Each Conduit.DLL reads from (or write to) this area in its own specific data formats.

It is usually desirable to keep the allocating and freeing responsibilities in the same component of code. Because of this, the following rules on dynamically allocated memory are enforced:

- The calling Conduit.DLL must pre-allocate memory onto the `m_pBytes` pointer before invoking any SyncManager APIs which use these structures
- The SyncManager does not allocate any memory into these structures and is also not responsible for freeing any memory within the structures. Sole responsibility lies within the calling client Conduit.DLL

Structures with Dynamically Allocated Memory:

- CRawRecordInfo - used by the record-oriented API
- CDbGenInfo - used by the file-oriented API
- CPositionInfo - used to obtain record position information

Conduits and the Windows Registry

When HotSync synchronizes all device applications, it relies on information on the PC to find each application. All conduits must enter the following information in the Windows Registry (Windows NT/Windows 95) or an HSM11.ini file:

- **Required information for HotSync**—When HotSync is first started, it reads the Windows Registry to find the conduits it needs to load and execute. This plug-in architecture allows for easy configuration of a user's PC for new or updated conduits.

- **Optional information for HotSync**—You can place information in the Registry that HotSync needs to execute or communicate to a conduit

HotSync loads only conduits named in the Windows Registry under one of the following keys:

- Key for Pilot applications- HKEY_CURRENT_USER\Software\Palm Computing\/ Pilot Desktop\Component*X*
- Key for add-on conduits- HKEY_CURRENT_USER\Software\Palm Computing\/ Pilot Desktop\Application*X*

Naming Third Party Conduits

The desktop software that ships with Pilot contains four native conduits, which are named starting with the Component0 Registry key.

To keep third party conduits separate from the native conduit entries, place them under the Application*X* key where

- Application*X* is a sequentially numbered entry representing the first third party conduit (e.g., Application0)
- Application1 represents the second conduit

and so on. The Pilot Desktop software does not need to be installed in order for HotSync and a third party conduit to function. However, there are some reserved Registry entries that cannot be used by third parties.

Registering Third Party Conduits

When you install your conduit for the first time, you have to register it as part of the installation process. This includes the following:

- Providing the Conduit Name
- Providing Name/Data Pairs

Providing the Conduit Name

HotSync expects that conduits are named in sequential order. The conduit name (Application*X*) therefore needs to be based on the number of the conduit that was last loaded. If the last conduit loaded was named Application5, your conduit needs to be named Application6.

Note that if you are supplying a de-installation procedure with your conduit, you need to be sure that all conduits loaded after it are renamed to maintain the proper numbering sequence. You may archive this either by changing the name of the last conduit to have the name of the deleted conduit or by changing the name of each conduit. The important issue is that the sequence of numbers is not interrupted.

Providing Name/Data Pairs

A set of required name/data pairs is under the Application*X* key. These name/data pairs describe the instructions that HotSync gathers as it scans through the Windows Registry at startup. Some of the information is used only by HotSync, while others, for example File0, are only passed along to the target conduit when HotSync loads it into memory. Because a single conduit synchronizes one database on the device with one database (or file) on the PC by default, every third-party conduit requires a new Application*X* Registry key and its required set of name/data pairs.

The following table lists the minimum set of name/data pairs placed under an Application*X* registry key.

Name	Type	Value (Address Book Application)
Conduit	String	addbook.dll
Creator	DWORD	0x61646472 ('addr')
Remote0	String	AddressDB
Directory	String	addbook
File0	String	*databaseName*.db
Integrate	DWORD	0
Module	String	foo
Name	String	Address Book
Priority	DWORD	2

The following table provides descriptions of name/data pairs under an Application*X* key.

Name	Data
Conduit	String indicating the disk filename of the third party conduit DLL. This disk file needs to be placed somewhere within the PATH environment variable. Conduits are generally installed in the same directory as HotSync.
Creator	Hexadecimal numeric value matching the Creator ID of the application residing on the Palm OS device. This value allows creation/modification of the remote database. This unique key ties the Palm OS application's database to a conduit on the PC.
Remote0	String indicating the name of the database residing on the device. This is a case-sensitive string and it is used by the native conduit logic in its remote File Open activities.
Directory	A string indicating the local PC directory to be created under the username directory. This directory may hold support files needed to accurately perform a record-level synchronization with a third party database, such as record ID mapping files. This directory will be the current directory when the conduit is invoked.
File0	A string indicating the local PC directory and filename of the third party database (or file) to be synchronized with the Pilot database named in the Remote0 Name/Data pair. This file is usually in the above-named directory.
Integrate	A hexadecimal value that for third party entries. Should be set to 0.
Module	A string which for third party entries can contain anything. This string is not used, but must be present.

Name	Data
Name	A string which is displayed in the HotSync Progress dialog to identify which conduit is currently executing.
Priority	Indicates the execution priority. Conduits with lower numbers execute before higher ones. Minimum value is 0, maximum is 4. If two conduits have the same priority, their execution order is undefined. Defaults to 2 if you don't provide a value. Don't change the default unless your conduit relies on a certain execution order. In the example below, the ToDo native conduit is assigned a priority of 1, which would cause it to execute before other native conduits.

Registry Entry Example

```
[HKEY_CURRENT_USER\Software\Palm Computing\/
Pilot Desktop\Component2]
"Module"="todo.dll"
"Conduit"="todcn11d.dll"
"Creator"=dword:746f646f
"Directory"="todo"
"File0"="todo.dat"
"Remote0"="ToDoDB"
"Priority"=dword:1
```

Default Registry Keys

To operate correctly, HotSync needs other Registry keys in addition to the ApplicationX Registry keys. These Registry keys do not need to be entered by a third party conduit author. HotSync can set up a set of default Registry keys that allow it to operate normally. HotSync will set up all of the default Registry keys (excluding any ApplicationX keys) when invoked with the command line argument -r:

```
HotSync -r
```

Important: Execute this command only once to initialize the Windows Registry. Any subsequent invocations (including the -r switch) overwrites existing Registry keys with the default values.

If a third party conduit requires any additional custom information when it executes, you may place additional name/data pairs under its Application*X* key, as long as the mandatory pairs are presented first.

Installing and Removing Your Conduit

This section discusses Installing Your Conduit and Removing Your Conduit:

Installing Your Conduit

When the end-user first installs your application, it must install your conduit and also HotSync1.1 and the 1.1 Pilot Desktop conduits.

- HotSync 1.1 Installation is required because HotSync 1.0, which is included with the Pilot package, was compiled with VC++ 2.2 MFC Library 2.2. Because most developers are now using VC++ 4.0 and MFC Library 4.0, it's necessary you include HotSync 1.1 in your package.
  All the files you need to install HotSync 1.1 are included with the Conduit SDK. The steps are described in some detail in HotSync 1.1 Installation.
- Conduit Installation is required so HotSync knows your application's key and registry information.

HotSync 1.1 Installation

1. Make sure HotSync is not running.
2. Copy the HSMII.EXE from poscond\condsdk\bin to the Pilot Desktop directory.
3. Copy the 1.1 libraries from poscond\condsdk\bin to the Pilot Desktop directory:
   – TABLE11.DLL
   – CMDS11.DLL
   – SYNC11.DLL

- BAKCN11.DLL
- INSCN11.DLL
- PDN11.DLL
- PDCMN11.DLL

4. Copy the 1.1 conduits to the Pilot Desktop directory:
   - ADDCN11.DLL
   - DATCN11.DLL
   - MEMCN11.DLL
   - TODCN11.DLL
5. For Windows 3.1 installation, edit hsm11.ini OR.
6. For Windows95 or Windows NT installation, make the following changes to the registry:

   This may be automated using a registry extract file. See the release notes for more information.

Action	Component	From	To
Change	Component0 conduit Value	Conduit = datacond.dll	Conduit = datacn1.1.dll
Change	Component1 Conduit Value	Conduit = addcond.dll	Conduit = addcn11.dll
Change	Component2 Conduit Value	Conduit = todcond.dll	Conduit = todcn111.dll
Change	Component3 Conduit Value	Conduit = memcond.dll	Conduit = mamcn11.dll
Change	HotSync Manager Back-upConduit Value	BackupConduit = bakcond.dll	BackupConduit = bakcn11.dll
Change	HotSync Manager In-stallConduit Value	InstallConduit = instcond.dll	InstallConduit = inscn11.dll
Add	HotSync Manager Notifier0 Value		Notifier0 = pdn11.dll

Conduit Installation

After you've successfully installed HotSync1.1 and modified the registry appropriately, you can install your conduit as follows:

1. Use `GetProfileInt` to look for the first open Application*X* key, starting with Application0. Increment until you've found the last application (Application*N*).
2. Add 1 to that number and add your application to the registry as (Application*N* + 1) using `SetProfileInt`
3. Add the standard name/data pairs for your application to the registry using `SetProfileInt` (see Providing Name/Data Pairs)

NOTE: You have to restart HotSync at this time and at any other time you've made changes to the registry (or .INI file).

Removing Your Conduit

It is customary that applications provide end-users a facility to easily remove all the relevant files and other information and restore the conduits for Pilot Desktop if necessary.

If you choose to do so, your application should follow these steps:

1. Remove the Application*X* key for your application from the registry
2. Decrement all Application*X* keys that follow to eliminate any gaps; HotSync relies on consecutive numbering of applications for execution

Cable vs. Modem Connection

Synchronization between PC and device applications can take place via a cable attached to a serial port on the PC or via a modem. Before requesting synchronization, the user must indicate cable or modem connection by selecting the appropriate command from the HotSync menu in Pilot Desktop. The selection is recorded in the Windows Registry.

When HotSync is started, it checks the Registry to determine whether a cable or modem was selected. It then opens the appropriate communications port to allow synchronization through the cable or modem.

In most cases, all other steps in the synchronization process are identical regardless of whether cable or modem connection is specified, and there is no impact on the conduit or the function calls needed for synchronization.

FastSync and SlowSync

If your conduit takes advantage of the native synchronization logic, it can perform two different types of synchronization.

The HotSync application generates a recommendation of which type to use. If your conduit does not use the native sync logic, this information can be ignored.

FastSync and SlowSync are two types of record-level synchronization. When the user starts the HotSync process, HotSync determines whether to perform a FastSync or a SlowSync. This decision is based on the last PC ID, which is stored on Pilot. A SlowSync is performed if this ID does not match the PC on which HotSync is currently executing, that is, if Pilot was last synchronized with a different PC than the one currently being used.

The HotSync synchronization decisions (including FastSync or SlowSync) are packaged into the `CSyncProperties` structure, which is passed to each conduit when it is started.

- **FastSync**.The record status fields must be accurate in order for data to be synchronized properly in a FastSync. To optimize the synchronization process, only records that have been modified since the last synchronization are retrieved from Pilot; records that have not been modified do not get retrieved from Pilot. In most cases, if a corresponding PC record is found for a Pilot record flagged as modified, the records are compared and record-level synchronization is carried out.
- **SlowSync**. If a user wants to synchronize one Pilot with two different PCs, the record status fields on the Pilot are cleared after synchronizing with the first PC, and will be cleared after the user synchronizes their Pilot with the second PC. In this scenario, a SlowSync is required.

  In a SlowSync, every record is retrieved from Pilot for comparison. Since the status fields have been cleared after the previous synchronization, they cannot be used to detect modifications after the Pilot has been syncronized with a different PC. During a SlowSync, corresponding records from Pilot and the Pilot Desktop are compared, and record-level synchronization is carried out.

To determine whether Pilot records with a status field of 0 have been modified since the last synchronization, HotSync searches for the record ID in the PC backup file (before the last sync).

– If the record exists in the backup file, the records are compared to determine whether the record has been modified since the last sync.

– If the record has been modified, its flag is set to `Modify`, and record-level synchronization proceeds.

3

Conduit Design Decisions

The minimal conduit structure described in Conduit Basics allows your conduit to provide a wide range of functions, including import/export, transaction processing and mirror-image synchronization. This structure allows conduits to be written in either C or C++.

This SDK provides two samples of simple C conduits, which use the SyncManager API to import and export Pilot Memo data to a text file on the PC (Txtcond, step 1, and step 2).

The SDK also provides source code for the four main conduits of Pilot Desktop (Memo, Date Book, Address Book and To Do List). These conduits are written in C++ and perform a very complex record-level synchronization with the PC that results in a perfect mirror-image of the data on the PC and the Palm OS device. All user changes on either side will be propagated to both sides during this process. These conduits also archive deleted records to separate files on the PC at the user's request.

In order to decide which sample conduit to use as a model for your conduit, answer the following question:

> Will your conduit be performing a record-level mirror-image synchronization between the two devices?
>
> – If not, the simple text conduit examples are the best to follow
>
> – If it is, the C++ examples might be the best to follow

This chapter explores several issues you need to consider when deciding whether to use the existing C++ conduit sample or to write your own. It discusses these topics:

- Conduit Design Questions
- Using the Native Synchronization Logic
- Record-Level Synchronization with Pilot Applications
- Pilot Desktop OS Native Synchronization Algorithm

Conduit Design Questions

You should note that it is possible to perform record-level, mirror-image synchronization with a straight - C conduit using only the Sync Manager API calls, but you can save a great deal of time and effort writing this sort of conduit by using one of our examples.

Consider the following issues which indicate the extent of the modifications that one of the synchronization samples requires:

- The samples sync with the four main Pilot ROM applications. If your conduit syncs with another Pilot application, you will have to change the data conversion routines between the Pilot and the conduits main data structure (a C Base Table subclass).
- The samples depend on unique record ID's on both devices when locating records for comparison. Unique ID's should always be present in the Pilot data. If the PC data does not have unique ID's, another unique key will have to be used for locating and comparing records.
- You will most likely have to map the record ID's of the Pilot to the record ID's of the PC application as records are read in from the PC's data files. Developers generally choose to store their Pilot-to-PC mappings in a text file in the user's Pilot directory on the PC.
- The sample conduits make extensive use of the status flags associated with each Pilot record (Deleted, Changed . . .). Is similar information available for the PC data? If it is not, then a copy of the PC data from the previous sync may have to be kept on the PC to do comparisons against.
- The sample conduits synchronize the Pilot's categories with the PC. Does the PC data have categories? Are they modifiable? How well do they map to the categories on the Pilot? Is there an alternative way to map the data without losing the category information? The functions of the Category Manager object may have to be modified to handle these differences.
- The sample conduits perform two types of synchronization—FastSync and SlowSync.

  The FastSync is highly optimized to limit the amount of data that must be transmitted between two devices. It makes extensive use of the modification status flags on both sides.

  The SlowSync does not use the status information. All Pilot records are copied to the PC and compared with their PC counterparts to determine modifications.

The HotSync manager will determine which type of sync is appropriate based on the Pilot's status information. If the Pilot was previously synchronized with a different PC, its status flags would have been cleared and are therefore not accurate with respect to the current PC.

Your conduit may or may not elect to handle these two variations. You may opt to use the same logic for both.

To determine whether you choose to base your conduit on one of the C++ record-level synchronization conduits, or the simpler C conduits, consider the following issues:

- All of the sample conduits support a concept known as archiving records. When a user deletes a record on the Pilot he may opt to keep an archive copy on the PC. These conduits will recognize this and will copy the record to an archive file on the PC before completing the deletion. Your conduit may or may not provide this function.
- Is there a defined API for accessing data in the PC files? Is it implemented in C or C++? This will affect the architecture of your conduit.
- Is your conduit going to synchronize/import/export only a subset of the data on the Pilot or PC?

  If so, consider using categories on the Pilot to define that subset, and consider whether or not you want to reset the modification status flags for all of the records on the Pilot at the conclusion of the sync.

  Also, consider how you will keep track of which subsets of data are currently on each device.
- Is your conduit going to access data from multiple PC files/applications or multiple Pilot databases/applications?

  If so, consider whether the order of conduit execution will affect the reliability of the data in question. In general, we don't recommend that one conduit attempt to sync the data of multiple applications.

  If your conduit accesses the data of other applications it should be either read-only, or for the purposes of updating cross-reference links between two applications. In the latter case, you should insure that your conduit runs before that of the other application. See Conduit Execution Order.
- Note that your conduit will be executed for each user/Pilot that HotSync's with the PC where the conduit is installed. HotSync does not allow different sets of conduits to be specified for different us-

ers. Therefore, your conduit should check the user name field passed with the SyncProperties structure and use it to decide which PC data to sync with, if any.

- When synchronizing two applications (PC + Pilot) whose fields do not match exactly, great care should be taken in mapping the fields to one another. This is the most critical and user-noticeable design decision you will make in your conduit development. Is data loss acceptable when fields on one side don't have counterparts on the other side? If not, the following two strategies have been used successfully:
  - Hide the extra data fields on one side in the note field on the other side
  - Cache the extra fields (from either or both sides) in the same file used for record ID mapping on the PC

Using the Native Synchronization Logic

To develop a conduit that uses the synchronization logic provided by the C++ classes the four native applications use, the conduit must meet the following requirements:

- Be constructed as a DLL with a single known "C" entry point
- Be written in C++
- Derive a class from CBaseMonitor to control overall sync logic and Implementing a Conduit. See Control Flow of Pilot Desktop's Native Synchronization Logic for details.
- Convert its own record formats into a CBaseRecord subclass using subclasses of CBaseTable, CBaseLinkConverter, CBaseSchema and CBaseIterator objects
- Rely on HotSync and it's libraries for all serial communications

The following sections provide additional information:

- Pilot Desktop OS Native Synchronization Algorithm
- Record-Level Synchronization with Pilot Applications
- Archiving Records

Pilot Desktop OS Native Synchronization

Algorithm

Most of the default synchronization logic for the sample conduits resides in BaseMon.cpp. This default synchronization will produce identical data on both platforms (the PC and the Pilot) at the end of the sync.

The default synchronization logic assumes that each PC and device application record has a `status` field to indicate that one of the following conditions has occurred since the last synchronization:

- The record has not been modified (No Modify)
- The record has just been added (Add)
- The record has been modified (Modify)
- The record has been deleted (Delete)
- The record has been archived (Archive)

Note: Archive means to save the record in an Archive file and remove the record from the current platform.

The conduit compares Pilot records with records in the PC table, and takes an action based on the status of each record. The following table summarizes the possible synchronization cases and describes the action taken to synchronize the records.

Pilot	PC	Action
Add	No Record	Add the Pilot record to the PC.
No Record	Add	Add the PC record to Pilot.
Delete	No Modify	Delete the record on Pilot and the PC.
No Modify	Delete	Delete the record on Pilot and the PC.
Delete	Modify	Instead of deleting the Pilot record, replace the Pilot record with the PC record. Message is sent to the log.
Modify	Delete	Instead of deleting the PC record, replace the PC record with the Pilot record. Message is sent to the log.
Modify	No Modify	Replace the PC record with the Pilot record.

Pilot	PC	Action
No Modify	Modify	Replace the Pilot record with the PC record.
Modify	Modify	If changes are identical, no action is taken.
Modify	Modify	If changes are different, add the Pilot record to the PC, and add the PC record to Pilot. Message is sent to the log.
Archive	No Record/ No Modify	Archive the Pilot record. If the PC record exists, delete it.
Archive	Delete	Archive the Pilot record. Delete the PC record.
Archive with No Modify	Modify	Instead of archiving the Pilot record, replace the Pilot record with the PC record. Message is sent to the log.
Archive after Modify	Modify	If the records are identical, archive the Pilot record and delete the record from the PC.
Archive after Modify	Modify	If changes are different, do not archive the Pilot record. Add the Pilot record to the PC and add the PC record to Pilot. Message is sent to the log.
No Record/ No Modify	Archive	Archive the PC record. If the Pilot record exists, delete it.
Delete	Archive	Archive the PC record. Delete the Pilot record.
Modify	Archive with No Modify	Instead of archiving the PC record, replace the PC record with the Pilot record. Message is sent to the log.
Modify	Archive after Modify	If the records are identical, archive the PC record and delete the record from Pilot.
Modify	Archive after Modify	If changes are different, do not archive the PC record. Add the Pilot record to the PC, and add the PC record to Pilot. Message is sent to the log.

Record-Level Synchronization with Pilot

Applications

To perform record-level synchronization with the four native applications on the first Pilot device, the third-party database schema should meet the following guidelines:

- A unique key (usually a record ID) must be present in each record of the database. If the unique key of a record is a user-editable field, or combination of fields, comparisons will be somewhat less reliable.

  Unique ID's for Pilot records should be assigned by the Pilot application. To do this, a new record is passed to the Pilot (via a Sync-Manager call) with an ID of 0. The Pilot will return the assigned ID number.
- A one-to-one relationship must exist between individual records in both databases.

  While it may seem obvious when thinking about an address database (for instance), this becomes an issue when dealing with the Pilot's Date Book database. The Pilot Date Book stores all repeating event information in a single physical database record, as it does with non-repeating events. However, some Desktop PIMs produce multiple physical database records when they store repeating event data. This makes record-level synchronization much more difficult.

Archiving Records

For the sample Pilot Desktop conduits, records that are marked 'archive' are placed in the appropriate archive file depending on the application and category. The archive filename is derived from the category name and application extension. For example, an archived Address Book record under the Unfiled category would be saved in a file called UNFILED.ABA. All archived records, whether they originate from Pilot or the PC, are stored on the PC. Archive files can be read into the desktop application using the `OpenArchive` command.

During synchronization, after the records marked to be archived are added to the PC Archive file, they are deleted from their current platform.

Consider using this feature or a similar one in your conduit. It can be a simple way of segmenting the data between the PC and Pilot.

We strongly recommend implementing this feature if your conduit sync's with one of the standard Pilot applications. Because archiving is a feature

of the standard Pilot applications, users won't understand it if it is not supported in the conduit.

4

Control Flow of Pilot Desktop's Native Synchronization Logic

This chapter examines the default synch behavior provided by the conduits of the PIM applications included with the first release of Pilot Desktop. Source code for these conduits is included as part of your Conduit SDK. To decide whether it makes sense for you to adapt one of these existing conduits to fit your application, see Chapter 3, Conduit Design Decisions. To understand more clearly what you have to do to implement your own conduit, look at Chapter 5, Implementing a Conduit.

This chapter discusses the following topics:

- Basic Control Flow
- Functions Called During Synchronization
- Synchronizing with Existing PC Applications
- Synchronizing Categories

Basic Control Flow

When the user presses the HotSync button on the cradle, the system goes through the following steps:

1. HotSync loads the tables library and instantiates
   – a `LocalTable` object that holds all application records stored on the PC

   – a `RemoteTable` object into which Pilot records will be loaded one at a time for processing.

2. HotSync calls the C entry point which in turn calls `OpenConduit`, passing in a `SyncProperties` structure.
3. From then on, a subclass of `CBaseMonitor` (the monitor) is in charge of the control flow. It iterates through all records in the table by calling `SyncGetNextModifiedRecord`.
4. The `SyncGetNextModifiedRecord`. routine calls the `Link-Converter` object to
   – Convert the Pilot record into the monitor object's common format
   – Place the converted record into the `LocalTable` object
5. The monitor object compares the record that was just loaded into `RemoteTable` with the records in the `LocalTable`.
   – If there is no record in the `LocalTable` that matches that from the `RemoteTable`, it creates a new record in `LocalTable` that will later be saved on the PC when the table is saved.
   – If there is a record with a matching ID, it compares the status of the two using the synchronization algorithm.

Note sync logic is stored only in the following locations:

- A class derived from `CBaseMonitor` contains all the synchronization logic (see Creating a CBaseMonitor Subclass)
- A class derived from `CBaseDTLinkConverter` performs PC to Pilot data conversion (categories & records) in both directions (see Creating a CBaseDTLinkConverter Subclass)
- A class derived from CBaseTable handles adding and removing records and record locating

Functions Called During Synchronization

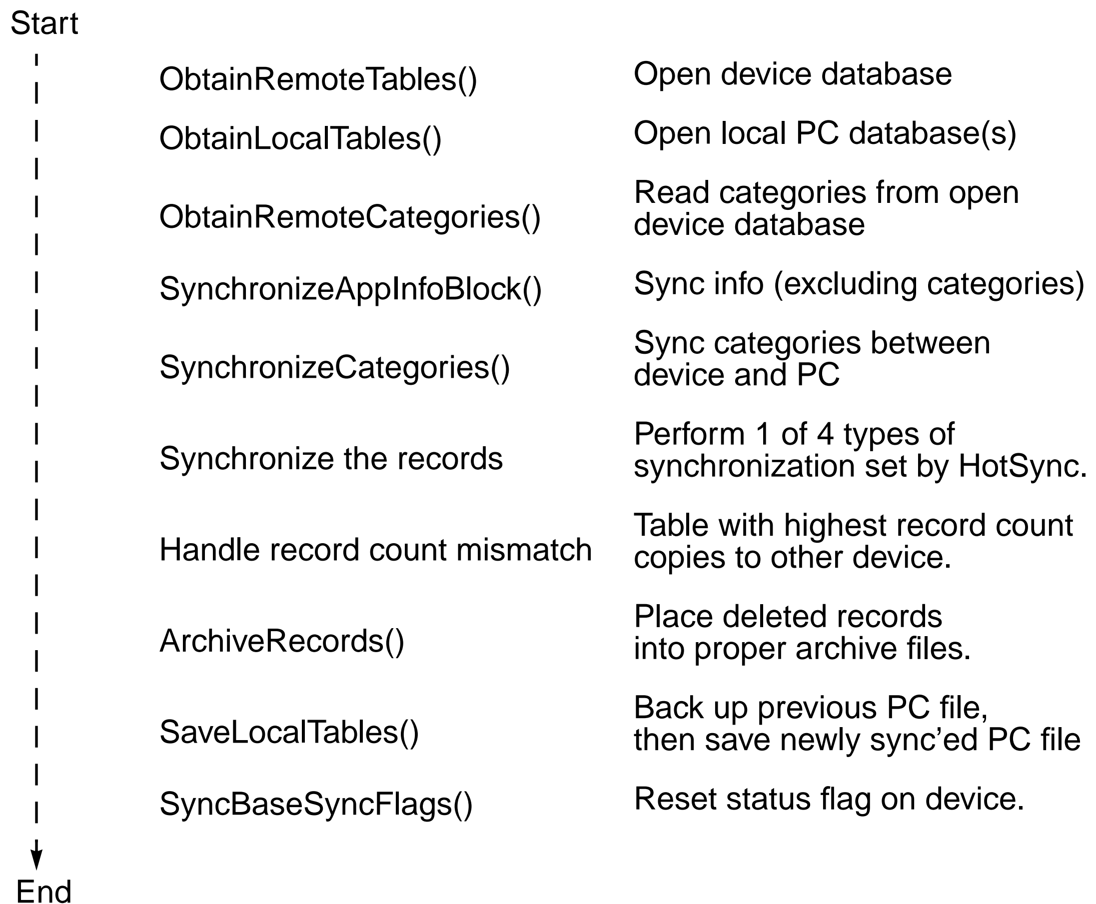


Figure 4.1 Functions Called During Synchronization

Synchronizing with Existing PC Applications

If you have a PC application that you want to synchronize and you want to use the Pilot Desktop's native synchronization logic, be aware that the native logic expects the following:

- A maximum of 16 categories total (PC and device combined); the category "unfiled" and 15 additional categories.
- That the record ID is assigned by the device, not the PC.

If your two databases don't meet these requirements, you need a preprocessor to do some work before calling the link converter to do its conversion.

You might also need a postprocessor to do some work after the link converter has done its conversion.

Synchronizing Categories

Categories are a central data handling concept for all Palm OS applications. Pilot Desktop's native logic synchronizes categories first. For more information, see Considering Category Manager Modifications before starting on the records.

5

Implementing a Conduit

This chapter helps you develop a conduit that's based on the C++ classes that provide the synchronization logic used by the native PIMs. It looks in some detail at each of the following steps a developer must take to implement such a conduit:

- Providing "C" Entry Points
- Creating a CBaseMonitor Subclass. Includes supplying a constructor that calls the base class constructor, and overriding the mandatory virtual member functions. It may also include overriding additional virtual functions.
- Creating a CBaseDTLinkConverter Subclass. Includes determining the need for a new converter class, writing the code for a new converter, and placing it in a separate source code file if necessary.
- Creating a CBaseTable Subclass Includes creating subclasses associated with `CBaseTable`, overriding some of the virtual functions, and incorporating necessary information in the header file.
- Considering Category Manager Modifications While many applications find they can use the native category behavior, you need to understand how categories are synchronized to decide whether you can use the native behavior.

Note that in this exploration of adapting the native logic to your application you will often encounter discussions of classes provided by the library that implements the native logic. These classes always have the word "Base" in them (e.g. `CBaseTable`). Their subclasses can have names you choose, it's probably best you replace "Base" with a word of your choice.

Providing "C" Entry Points

Every conduit must publish a `DllMain()` routine for the to be used by the Windows operating system and three public "C" entry points (`OpenCon-`

duit, GetConduitName, and GetConduitVersion) to be invoked by HotSync. They are discussed in some detail in this section. Place the code for all four functions in a single C++ source file, such as MY-COND.CPP.

Listing 5.1 C Entry Points

```
extern "C" {
typedef long (*PROGRESSFN) (char*);

ExportFunc long OpenConduit
                (PROGRESSFN, CSyncProperties&);
typedef long (*POPENCONDUIT)
                (PROGRESSFN, CSyncProperties&);

ExportFunc long GetConduitName(char*, WORD);
typedef long (*PGETCONDUITNAME) char*, WORD);

ExportFunc DWORD GetConduitVersion()
typedef DWORD (*PGETCONDUITVERSION)();
};
```

The rest of this section discusses:

- Providing a DllMain Routine. This function has to save the hInstance parameter which is needed by the class derived from CBaseMonitor.
- Providing Entry Point Routines. Three functions must be provided, OpenConduit, GetConduitName, and GetConduitVersion.

Providing a DllMain Routine

The DllMain routine is a typical 32-bit Windows DllMain routine, except for the saving of the passed in hInstance parameter. The CBaseMonitor that determines the control flow needs this instance handle. The Windows operating system automatically calls DllMain when it loads the conduit DLL into memory as a result of HotSync performing a LoadLibrary call on it.

Listing 5.2 DllMain Startup Routine

```
// Filename: mycond.cpp
// Description: Source code for the Windows
// DllMain() function and the Conduit routine
//'Open Conduit()'.

// Init global variable to null
HINSTANCE myInst = 0;
extern "C" int APIENTRY
DllMain (  HINSTANCE hInstance,
           DWORD dwReason,
           LPVOID lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        TRACE0("ADDCOND.DLL Initializing!");

        //Extension DLL one-time initialization
        AfxInitExtensionModule(  addcondDLL,
                                 hInstance);

        //Insert this DLL into the resource chain
        new CDynLinkLibrary (addcondDLL);

        myInst = hInstance;
      }
      else if (dwReason ==DLL_PROCESS_DETACH)
      {
        TRACE0("ADDCOND.DLL Terminating!");

      //properly clean up the extension module
      AfxTermExtensionModule(addcondDLL);
    }
    return 1        /ok
}
```

Providing Entry Point Routines

Your Conduit should provide 3 entry point routines, discussed in this section:

- The OpenConduit Function
- The GetConduitName Function
- The GetConduitVersion Function

The OpenConduit Function

When HotSync calls `OpenConduit`, the conduit carries out all its synchronization duties before returning control to HotSync. HotSync invokes this routine only once, immediately after it has dynamically loaded the conduit into memory.

The function is passed two parameters:

- A pointer to a callback routine within HotSync that a conduit can invoke periodically during its activities.
- A pointer to the `CSyncProperties` object which contains the characteristics of the current synchronization session (see Listing 5.4).

Because this function resides in the same source code file as `DllMain`, the `myInst` variable, which is set by `DllMain`, is accessible (see Providing a DllMain Routine.)

Listing 5.3 shows the `OpenConduit` function for the address book conduit included with your Conduit SDK.

Listing 5.3 OpenConduit Function

```
ExportFunc long OPENCONDUIT (
                        PROGRESSFN pFn,
                        CSyncProperties& rProps)
{
  long retval = -1;
  if (pFn)
    {
    CAddressConduitMonitor* pMonitor;
    pMonitor = new CAddressConduitMonitor(
                            pFn, rProps, myInst);
```

```
            if (pMonitor)
            {
                retval = pMonitor->Engage();
                delete pMonitor;
            }
        }
        return(retval);
}
```

The two parameters passed in by HotSync (callback routine and CSyncProperties instance) are passed into the constructor of the CBaseMonitor, along with the instance handle from DllMain.

CSyncProperties is a C++ class, however all of its members are public. As a result, using it's similar to using a traditional C structure. CSyncProperties contains:

- Much of the vital information regarding the nature of the synchronization process to execute.
- Assisting information (such as filenames) for synchronization of the local and remote databases.

Other data members serve as function parameters for some of the SyncManager function calls that a conduit must invoke to control the Palm OS device. HotSync supplies all the information inside CSyncProperties.

Listing 5.4 CSyncProperties class

```
enum eSyncTypes { eFast, eSlow, eHHtoPC, ePCtoHH,
                  eInstall, eBackup};
enum eFirstSync { eNeither, ePC, eHH};
enum eConnType  { eCable, eModem};

class CSyncProperties {
public:
    eSyncTypes m_SyncType;
    char       m_PathName[256];
    char       m_LocalName[256];
    char       m_UserName [256];
```

```
    char*      m_RemoteName[DB_NAMELEN];
    CDbList*   m_RemoteDbList[DB_NAMELEN];
    int        m_nRemoteCount;
    CSyncLog*  m_pSyncLog;
    DWORD      m_Creator;
    WORD       m_CardNo;
    DWORD      m_DbType;
    DWORD      m_AppInfoSize;
    DWORD      m_SortInfoSize;
    eFirstSync m_FirstDevice;
    eConnType  m_Connection;
    char       m_Registry[256];
    HKEY       m_hKey;
};
```

The GetConduitName Function

This function is the extern "C" entry point into the conduit which returns the name to be used when displaying messages regarding this conduit:

Listing 5.5 GetConduitName Example

```
ExportFunc long GETCONDUITNAME(char* pszName,
                                WORD nLen)
{
  long retval = -1;
  if (::LoadString( myInst, IDSTR_ADDRESSBOOK,
                    pszName, nLen))
      retval = 0;
  return retval;
```

The GetConduitVersion Function

This routine is the extern "C" entry point into this conduit which returns the conduits version number.

Listing 5.6 GetConduitVersion Example

```
ExportFunc DWORD GETCONDUITVERSION()
{
  return ADDRESS_CONDUIT_VERSION;
}
```

Creating a CBaseMonitor Subclass

This section looks in some detail at the `CBaseMonitor` class. This class determines the control flow for the conduit and is at the heart of the native synchronization logic. Its data members store information about the synchronization process; its member functions determine the behavior. Every conduit has to create a subclass of this class because it's necessary to override a number of virtual functions that by default have no behavior.

You may decide to use or adapt some of the virtual functions defined by the conduits of the four native applications (which are included in the SDK), but you cannot use the class as is because the functions have no behavior at that level.

You learn about the following aspects of `CBaseMonitor`:

- CBaseMonitor Basic Structure
- CBaseMonitor Data Members
- CBaseMonitor Functions Must to Override
- CBaseMonitor Functions You May to Override

CBaseMonitor Basic Structure

The `CBaseMonitor` class provided in Basemon.h contains data members necessary for performing all required synchronization activities. When you derive a subclass from `CBaseMonitor`, you must initialize some of these data members in order for the base synchronization logic to execute successfully. Listing 5.7 is an excerpt of the class definition for `CBaseMonitor` from the header file BASEMON.H in the Conduit SDK; it shows the data members that need to be initialized.

Listing 5.7 CBaseConduitMonitor Class Data Members

```
//
// Base Monitor
//
class CBaseConduitMonitor {
protected:
    CBaseDTLinkConverter*  m_pDTConvert;
    PROGRESSFN             m_pfnProgress;
    CBaseTable*            m_LocRealTable;
    CBaseTable*            m_LocArchTable;
    CBaseTable*            m_BackupTable;
    CBaseTable*            m_RemRealTable;
    CSyncProperties        m_rSyncProperties;
    CCategoryMgr*          m_LocCategory;
    CCategoryMgr*          m_RemCategory;
    BYTE                   m_RemHandle;
    char                   m_ArchFileExt[5];
    int                    m_TotRemoteDBs;
    int                    m_CurrRemoteDB;
    CDbGenInfo             m_DbGenInfo;
    HINSTANCE              m_DllInstance;
```

The following data members are the most important ones:

- `m_pDTConverter` points to a converter object that converts record data obtained from the device into a format used by the monitor object's synchronization logic (see Creating a CBaseDTLink-Converter Subclass).
- `m_LocRealTable` is the table that will hold data resident on the PC.
- `m_RemRealTable` is the table that will hold data coming from the device.
- `CSyncProperties` (see Listing 5.4) is a copy of the object passed into the conduit by HotSync when it invokes the conduit.

Many of the member function prototypes define a parameter as a `CBaseRecord&`, then rely on the code you provide to cast the parameter to the

specific record object that the conduit is synchronizing. Because casting happens so low in the hierarchy, the core synchronization logic only has to deal with `CBaseRecord` instances. This helps reduce its compile time exposure to the growing list of header files containing class definitions for `CBaseRecord` subclasss. It also makes it possible to have all conduits use the same core logic.

CBaseMonitor Data Members

The following data members of `CBaseMonitor` are discussed below:

- CBaseDTLinkConverter* m_pDTConvert
- CBaseTable* m_LocRealTable
- CBaseTable* m_LocArchTable
- CBaseTable* m_BackupTable
- PROGRESSFN m_pfnProgress
- CSyncProperties m_rSyncProperties
- CCategoryMgr* m_LocCategory
- CCategoryMgr* m_RemCategory
- BYTE m_RemHandle
- CBaseTable* m_RemRealTable
- int m_CurrRemoteDB
- CDbGenInfo m_DbGenInfo
- HINSTANCE m_DllInstance

CBaseDTLinkConverter* m_pDTConvert

A converter object that is usually created from within the constructor for the `CBaseMonitor` subclass. The converter must understand the record layouts living on the Palm OS device; it has to transform that record information from device format into a format the synchronization logic can use.

Most conduits derive a class from `CBaseDTLinkConverter` to handle new file formats on the Palm OS device (see Creating a CBaseDTLinkConverter Subclass).

PROGRESSFN m_pfnProgress

HotSync supplies this function pointer. It allows the conduit to call back into HotSync and periodically report its progress. This function pointer

currently has no effect on HotSync, and is in place for possible future expansion. For now, the only requirement of a monitor subclass is to pass the first parameter of its constructor down to its base classes constructor, and ignore this data member.

CBaseTable* m_LocRealTable

This data member is a pointer to the table that contains all the records on the PC. The function `ObtainLocalTables` opens and reads in this table.

Your base monitor subclass must create an instance of your subclass of the `CBaseTable` class (see Creating a CBaseTable Subclass), that is then used to store the data retrieved by ObtainLocalTables.

NOTE: No synchronization can occur unless this data member is correctly initialized.

CBaseTable* m_LocArchTable

Represents an archive database on the PC; a file which can store all records from the main database the user marked for archiving. This table and `m_LocRealTable` have to be an instance of the same subclass. Initialize this table in the function CreateLocalArchTable.

CBaseTable* m_BackupTable

This data member and the `m_LocRealTable` data member must belong to the same class. This table represents a backup of the original PC database after the last synchronization. It provides a snapshot of the PC data as it looked after the last synchronization session ended.

A backup table is important when HotSync has to perform a slow synchronization. HotSync decides to perform a slow synchronization when it finds that no record status flags are set on the device’s database. This may occur if the device initiates a synchronization session with more than one PC because the built-in synchronization logic clears all the status flags at the end of a session, in preparation for detecting future record alterations.

CBaseTable* m_RemRealTable

This table object represents the database on the device that will be synchronized with its counterpart on the PC. This table and the

m_LocRealTable data member must be instances of the same subclass of CBaseTable because the native synchronization logic can't compare objects that aren't instances of the same class. This data member is initialized when the native synchronization logic invokes the ObtainRemoteTables virtual member function.

CSyncProperties m_rSyncProperties

This data member is a copy of the SyncProperties object passed from HotSync into a CBaseMonitor subclass constructor. Normally, the constructor of a CBaseMonitor subclass should simply pass the CSyncProperties parameter down to its base class constructor, where a copy is made into this data member.

CCategoryMgr* m_LocCategory

A pointer to the category manager that contains all the categories that exist on the PC.

CCategoryMgr* m_RemCategory

A pointer to the category manager that contains all categories that exist on the device.

BYTE m_RemHandle

If the SyncManager calls ObtainRemoteTables, and if the device successfully opens the named database during execution, a handle is returned by the function call. The returned handle should be stored in the data member m_RemHandle, because it's needed by several of the SyncManager functions.

char m_ArchFileExt[5]

Holds the PC file extension that is used when creating a local archive disk file. This data member may be populated within the virtual member function ObtainLocalTables. It should be a NULL-terminated string.

int m_TotRemoteDBs

Holds the number of remote databases to be opened during the current synchronization session. This is currently always set to 1 but provided in case a conduit has to open more than one remote database to synchronize correctly with a local PC database(s).

A limitation on the current Pilot device prevents more than one remote database to be open concurrently.

int m_CurrRemoteDB

Holds the current offset into an array of remote database names (zero-based). When a conduit is dealing with more than one remote database, HotSync hands it an array of database names within the `CSyncProperties` object.

NOTE: This data member must be set to 0 (zero) in the constructor of the `CBaseMonitor` subclass.

CDbGenInfo m_DbGenInfo

Used by SyncManager function calls as a convenience to the built-in synchronization logic. Subclasses of `CBaseMonitor` don't need to perform any actions on this data member.

HINSTANCE m_DllInstance

Used by the `CBaseMonitor` class for discovering strings from a resource file and for logging conflicts. The third parameter in the constructor, which represents the instance handle for the Conduit.DLL, must be passed down to the base class constructor. The keyword ExportFunc used in the OpenConduit function prototype is defined in the header file SYNCMGR.H and exports the OpenConduit function from the DLL. This eliminates the need to place it in the EXPORTS section of the module definition file.

CBaseMonitor Functions Must to Override

This section discusses the constructor and the virtual functions that you must override if you derive a class derived from `CBaseMonitor` as follows:

- Monitor Constructor and Destructor
- ObtainLocalTables
- ObtainRemoteTables
- AddRecord
- AddRemoteRecord
- ChangeRemoteRecord

- CreateLocalArchTable
- FastSyncRecords
- SlowSyncRecords
- CopyRecordsPCtoHH
- CopyRecordsHHtoPC
- LogRecordData
- LogApplicationName

By default, these virtual functions in `CBaseMonitor` do nothing. Unless the subclass supplies working code for them, the conduit fails.

NOTE: You can use the code in the Examples section, which is from the CAddressConduitMonitor subclass, as a template for the functions you write.

```
CBaseConduitMonitor(PROGRESSFN pFn,
                    CSyncProperties&,
                    HINSTANCE hInst = NULL)
~CBaseMonitor()
virtual long ObtainRemoteTables(void)
virtual long ObtainLocalTables()(void)
virtual long AddRecord( CBaseRecord& rFromRec,
                        CBaseTable& rTable)
virtual long AddRemoteRecord(CBaseRecord& rRec)
virtual long ChangeRemoteRecord(CBaseRecord& rRec)
virtual long CreateLocalArchTable(CBaseTable*&)
virtual long FastSyncRecords(void)
virtual long SlowSyncRecords(void)
virtual long CopyRecordsPCtoHH(void)
virtual long CopyRecordsHHtoPC(void)
virtual long LogRecordData( DBaseRecord& rRec,
                            char* fieldInfo)
virtual long LogApplicationName (char* appName,
                                 WORD, len)
```

Monitor Constructor and Destructor

Every subclass of CBaseMonitor that you create has to have a constructor and destructor. The destructor is a standard C++ destructor. An example for the constructor is provided below.

Prototype

```
CAddressConduitMonitor(  PROGRESSFN pFn,
                         CSyncProperties& rProps,
                         HINSTANCE hInst)
```

Parameters

pFn	Pointer to a function existing in HotSync.
rProps	Reference to a `CSyncProperties` object.
hInst	Instance handle of the DLL.

Purpose

Every class derived from `CBaseMonitor` must supply its own constructor. This constructor is called from the C entry point routine `OpenConduit` as part of the conduit start-up. This constructor has to pass all three parameters to the `CBaseMonitor` constructor. Additional responsibilities are:

- To construct a proper converter object and populate the data member `m_pDTConvert` with its address.
- To set the following data members:
  `m_TotRemoteDBs = 1`
  `m_CurrRemoteDB = 0`

Return Codes

None.

Example

```
CAddressConduitMonitor:: CAddressConduitMonitor (PROGRESSFN pFn,
                         CSyncProperties& rProps, HINSTANCE hInst)

                        : CBaseConduitMonitor(pFn, rProps, hInst)
{
    m_pDTConvert = new CAddressDTLinkConverter
                        (rProps.m_pSyncLog, hInst);
    m_TotRemoteDBs = 1;
    m_CurrRemoteDB = 0;
```

```
}
```

ObtainLocalTables

Prototype `long ObtainLocalTables(void)`

Parameters None.

Purpose Populate the three data members `m_LocRealTable`, `m_LocArchTable`, and `m_BackupTable`.

Description This function needs to:

- Populate the data member m_LocRealTable with an instance of a `CBaseTable` subclass (see Creating a CBaseTable Subclass). Once this has happened, the new conduit should open its local PC disk file and read the existing data into this object, setting it up for synchronization. It's not necessarily a problem if no disk file is available from which to read data.
- Create an archive table object and place it into the data member `m_LocArchTable`, which also is derived from the class `CBaseTable`. If your conduit does not support archiving, deleted records, this data member should remain set to NULL.
- Populate the data member `m_BackupTable`, which is used mainly by the slow sync logic.

Return Codes 0 = success

CONDERR_BAD_LOCAL_TABLES

Example

```
long CAddressConduitMonitor::ObtainLocalTables(void)
{
long retval= CONDERR_BAD_LOCAL_TABLES;
long lTblErr;
CString dataFile(m_rSyncProperties.m_PathName);
dataFile += m_rSyncProperties.m_LocalName;
dataFile += DATA_EXT;
```

```
// Create our local table object and open it's disk file.
m_LocRealTable = new CAddressTable();
if (m_LocRealTable) {
    retval = 0;
    if (m_rSyncProperties.m_SyncType != eHHtoPC)    {
        lTblErr = m_LocRealTable->OpenFrom(dataFile, 0);
        if (!(lTblErr == 0 || lTblErr == DERR_FILE_NOT_FOUND))
            retval = CONDERR_BAD_LOCAL_TABLES;
    }
}
  // Create our local archive table object
  if (!retval)
    m_LocArchTable = new CAddressTable();

  if (m_LocArchTable)
  {
    // Set Archive File Extension
    strcpy(m_ArchFileExt, ARCHIVE_FILE_EXT);

    // Create Backup table object
    if (m_rSyncProperties.m_SyncType == eFast ||
        m_rSyncProperties.m_SyncType == eSlow)
    {
      if ((m_BackupTable = new CAddressTable()) == NULL)
        retval = CONDERR_BAD_LOCAL_TABLES;
    }
  }
  else
    retval = CONDERR_BAD_LOCAL_TABLES;

  return(retval);
}
```

ObtainRemoteTables

Prototype `long ObtainRemoteTables(void)`

Parameters None.

Purpose Populate the two data members `m_RemRealTable` and `m_RemHandle`. The chief purpose of this function is to instruct the Palm OS device to open a particular database that will be synchronized with a local PC database. This routine should create a database if a failure of the open function indicates that none exists.

Return Codes 0 = success

CONDERR_BAD_REMOTE_TABLES

Example

```
long CAddressConduitMonitor::ObtainRemoteTables(void)
{
long  retval;
// Call into SyncManager.DLL to open the Remote database
retval =
SyncOpenDB(m_rSyncProperties.m_RemoteName[m_CurrRemoteDB], 0,
                                        m_RemHandle);

// Create remote dataBase, if it's not there (check sync type)
if (retval == SYNCERR_FILE_NOT_FOUND &&
                    m_rSyncProperties.m_SyncType != eHHtoPC) {
    CDbCreateDB dbInfo;
    memset(&dbInfo, 0, sizeof(dbInfo));
    dbInfo.m_Creator  = m_rSyncProperties.m_Creator;
    dbInfo.m_Flags    = eRecord;
    dbInfo.m_CardNo   = (BYTE)m_rSyncProperties.m_CardNo;
    dbInfo.m_Type     = m_rSyncProperties.m_DbType;
    strcat(dbInfo.m_Name,
           m_rSyncProperties.m_RemoteName[m_CurrRemoteDB]);
    if (!(retval = SyncCreateDB(dbInfo)))  {
        m_RemHandle = dbInfo.m_FileHandle;
    }
}
```

```
// Need a table to hold converted remote records (one at a time)
if (!retval)
     {
     if (!(m_RemRealTable = new CAddressTable()))
          {
          SyncCloseDB(m_RemHandle);
          retval = CONDERR_BAD_REMOTE_TABLES;
          }
     }
return( retval);
}
```

AddRecord

Prototype

```
long AddRecord ( CBaseRecord& rFromRec,
                 CBaseTable& rTable)
```

Parameters

rFromRec	Record to be added to Table object.
rTable	Table object that is to receive new record.

Purpose

To populate the `rTable` table object with a new record using the `rTable` record object). The main purpose of this routine is to cast the generic incoming parameters to the specific table object needed by the conduit. The routine is called by the `CBaseMonitor` generic synchronization logic, where it does not have the typing information necessary to deal with all possible `CBaseRecord` subclasses. This function relies on a member function of the `CBaseTable` class, which adds a new record then populates it with information passed into it. (See `AppendDuplicateRecord` of the `CBaseTable` class.)

Return Codes

0 = success

CONDERR_ADD_LOCAL_RECORD

Example

```
long CAddressConduitMonitor::AddRecord(  CBaseRecord& rFromRec,
                                         CBaseTable& rTable)
{
```

```
    long  retval=0;

    // Cast the parameters to our own specific object types
    CAddressTable& rToTable     = (CAddressTable&)rTable;
    CAddressRecord& rFromRecord = (CAddressRecord&)rFromRec;

    // Instantiate a new record object to represent the fresh row
    CAddressRecord toRec(rTable, 0);

    if (rTable.AppendDuplicateRecord(rFromRecord, toRec))
        retval = CONDERR_ADD_LOCAL_RECORD;

    return(retval);
}
```

AddRemoteRecord

Prototype `long AddRemoteRecord (CBaseRecord& rRec)`

Parameters rRec — Record to be added to the remote database.

Purpose Add a new record to the remote database and obtain the newly assigned unique record ID.

- Allocate enough memory for the device format record layout.
- Convert the passed-in table record to the format needed by the device. This is done by the ConvertToRemote function in the link converter.
- Use the SyncManager to send the device data to the Palm OS device.
- After the SyncManager call, obtain the new unique record ID assigned by the device and store it in the ID field of the passed-in record object.

This function is called by the `CBaseMonitor` generic synchronization logic, where it does not have the typing information necessary to convert the base record object to the specific record layout used by the device.

Return Codes 0 = success

CONDERR_ADD_REMOTE_RECORD

CONDERR_CONVERT_TO_REMOTE_REC

Example

```
long CAddressConduitMonitor::AddRemoteRecord(CBaseRecord& rRec)
{
CRawRecordInfo rawRec;
CAddressRecord &rLocRec = (CAddressRecord&)rRec;
long            retval  = CONDERR_ADD_REMOTE_RECORD;

memset(&rawRec, 0, sizeof(rawRec));
rawRec.m_FileHandle = m_RemHandle; // remote file handle
rawRec.m_RecId      = 0 ;
                               // Palm OS device assigns new RecId

// Allocate memory for rawRecord.m_pBytes
if (!AllocateRawRecordMemory(rawRec, ADDRESS_RAW_REC_MEM)) {
    // Convert record data for remote, upon return grab new
    //RecordId.
    if (!(retval = m_pDTConvert->ConvertToRemote(
                                        rLocRec, rawRec)))  {
          if (!(retval = SyncWriteRec(rawRec)))
              rRec.SetRecordId(rawRec.m_RecId);
     }
     else
          retval = CONDERR_CONVERT_TO_REMOTE_REC;
     // Free memory not needed anymore
     if (rawRec.m_TotalBytes > 0 && rawRec.m_pBytes)
          delete rawRec.m_pBytes;
}
return(retval);
```

ChangeRemoteRecord

Prototype `long ChangeRemoteRecord (CBaseRecord& rRec)`

Parameters rRec Record to be overwritten in the remote database.

Purpose

To alter an existing record in the remote database on the device. The record is located by its unique key, the record ID.

- Ask for the unique record ID present in the passed-in record object. (This will be used as a key to look up the matching record on the device.)
- Allocate enough memory to hold the converted record format.
- Call on the converter data member to do the actual data conversion from a base record object to the record layout acceptable by the remote database.
- Call the SyncManager function `SyncWriteRec` to send the data to the Palm OS device.

Return Codes

0 = success

CONDERR_CHANGE_REMOTE_RECORD

CONDERR_CONVERT_TO_REMOTE_REC

Example

```
long CAddressConduitMonitor::ChangeRemoteRecord
                                              (CBaseRecord& rRec)
{
     CRawRecordInfo rawRec;
     CAddressRecord &rLocRec = (CAddressRecord&)rRec;
     long  retval = CONDERR_CHANGE_REMOTE_RECORD;
     int   locRecId;

     memset(&rawRec, 0, sizeof(rawRec));
     rLocRec.GetRecordId(locRecId);
     rawRec.m_FileHandle = m_RemHandle;
     // remote file handle
     rawRec.m_RecId      = (DWORD)locRecId;
     // key used for record location

     // Allocate memory for rawRecord.m_pBytes
     if (!AllocateRawRecordMemory(rawRec, ADDRESS_RAW_REC_MEM))  {
        // Prepare record data for remote, upon return grab
        // new RecordId.
```

```
          if (!m_pDTConvert->ConvertToRemote(rLocRec, rawRec))
               retval = SyncWriteRec(rawRec);
          else
               retval = CONDERR_CONVERT_TO_REMOTE_REC;

          if (rawRec.m_TotalBytes > 0 && rawRec.m_pBytes)
               delete rawRec.m_pBytes;
     }
     return(retval);
}
```

CreateLocalArchTable

Prototype `long CreateLocalArchTable (CBaseTable*& pBase)`

Parameters

pBase	Reference to a table pointer receiving allocated memory.

Purpose To create a conduit-specific table object to work with all archived records. The generic synchronization engine calls this virtual function when it's processing deleted records that optionally get stored in an archive database after removal from the main table. The archive table has to use the same schema as the main table (see CBaseSchema Class).

Return Codes 0 = success

-1 = could not allocate archive table object

Example

```
long CAddressConduitMonitor::CreateLocalArchTable(
                                             CBaseTable*& pBase)
{
     long retval = -1;
     pBase = new CAddressTable();
     if (pBase)
          retval = 0;

     return(retval);
```

```
}
```

FastSyncRecords

Prototype `long FastSyncRecords(void)`

Parameters None.

Purpose Perform an optimized record-level synchronization involving only the modified records from the Palm OS device. Each conduit has to supply this function because it has to create application-specific record objects used in traversing each of the two tables to be synchronized.

The function traverses the remote database and reads in records that have been modified since the last synchronization session. The device knows the modification status of a record; the status is available through the Sync-Manager function `SyncReadNextModifiedRec`. If no records have been modified since the last synchronization session, the SyncManger returns an end-of-file error on the first read and no more processing is necessary.

Once a remote record is obtained, the inherited base class routine `SynchronizeRecord` is invoked with the record. This routine contains all the native synchronization conflict resolution logic.

Return Codes 0 = Success

CONDERR_BAD_REMOTE_TABLES

CONDERR_CONVERT_FROM_REMOTE_REC

Example

```
long CAddressConduitMonitor::FastSyncRecords(void)
{
     long                retval = 0, err = 0;
     CRawRecordInfo      rawRecord;
     CAddressRecord      locRecord(*m_LocRealTable, 0);
     CAddressRecord      backRecord(*m_BackupTable, 0);

     memset(&rawRecord, 0, sizeof(rawRecord));
     rawRecord.m_FileHandle = m_RemHandle;
```

```
// remote file handle
 rawRecord.m_RecId      = 0;
// Palm OS device assigns RecId

 if (!m_RemRealTable)
      return(CONDERR_BAD_REMOTE_TABLES);
// Create record object to be a holding buffer for converted
// remote raw records. To store field values in a record
// object, our table object requires they be positioned in
// order.
CAddressRecord remRecord(*m_RemRealTable, 0);
if (m_RemRealTable->AppendBlankRecord(remRecord))
    return(CONDERR_BAD_REMOTE_TABLES);

// Allocate memory for raw record conversion buffer.
retval = AllocateRawRecordMemory(
                             rawRecord, ADDRESS_RAW_REC_MEM);

// The main loop iterating over the remote modified records.
while (!err && !retval)  {
    if (!(err = SyncReadNextModifiedRec(rawRecord))) {

        // Convert from raw record format to PC record object
        if (!m_pDTConvert->ConvertFromRemote
            (remRecord, rawRecord)) {

            // Call inherited base class function to
            // synchronize the record
            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);
```

```
    // Free memory allocated for raw record conversion buffer.
    if (rawRecord.m_TotalBytes > 0 && rawRecord.m_pBytes)
        delete rawRecord.m_pBytes;

    // Send all modified records to the Palm OS device. Supply a
    // record object to the inherited base class member function.
    if (!retval) {
        CAddressRecord loc2Rec(*m_LocRealTable, 0);
         retval = SendRemoteChanges(loc2Rec);
    }
    return(retval);
}
```

SlowSyncRecords

Prototype `long SlowSyncRecords (void)`

Parameters None

Purpose Applications use SlowSync when they can't rely on the status flags to be accurate. If the user has performed a HotSync with another PC, the status flags are cleared. SlowSync uses the backup file, which is a copy of the file after the last HotSync, to determine which records have been added, changed, or deleted on the device since the last HotSync. To perform a SlowSync, every record must be read in from the device. This contrasts with FastSync which reads only the modified records.

All the PC records have already been read into memory (into the `m_LocRealTable` table on the PC). For each PC record, if the statusFlag is None, it's set to Pending. Device records are read in one at a time.

Since the Palm OS device status flags may not be accurate, SlowSync proceeds as follows: If the device statusFlag is None, then that record is compared against the Backup file record to determine if the device record has been added, changed, or deleted. Then, each device record is compared with the record in the PC table (which contains the PC records along with the newly merged device records) to determine if the device record should be added to the PC table, replace the current PC record, cause the PC record to be deleted from the PC table, or be added to the Archive file. If the record exists on both the device and the PC and the PC record has a

Pending statusFlag, the statusFlag is changed to its appropriate value. This is important because in the second pass, if the statusFlag is Pending, that means that the record does not exist on the device and it does exist on the PC with no changes, therefore the record was deleted on the device so it needs to be deleted from the PC. After each device record has been read with its sync action performed to the PC table, then a second pass is made to the PC table. For each PC record that is marked as modified (statusFlag != None), a message will be send to the device to update that device record. After all the appropriate records are updated on the device and the statusFlags for each PC record have been cleared, then the PC table is ready to be written to the PC as the new PC file.

Example

```
long CAddressConduitMonitor::SlowSyncRecords(void)
{
  long              retval   = 0, tErr, err = 0;
  WORD              rawRecIx = 0;
  CRawRecordInfo    rawRecord;
  CAddressRecord    backRecord(*m_BackupTable, 0);
  CAddressRecord    locRecord(*m_LocRealTable, 0);
  CBaseIterator     locIterator(*m_LocRealTable);

  CString backFile(m_rSyncProperties.m_PathName);
  backFile += m_rSyncProperties.m_LocalName;
  backFile += BACK_EXT;

  memset(&rawRecord, 0, sizeof(rawRecord));
  rawRecord.m_FileHandle = m_RemHandle;    // Remote File Handle

  // Read in PC Backup file (file after last sync)
  tErr = m_BackupTable->OpenFrom(backFile, 0);

  if (!m_RemRealTable)
    return(CONDERR_BAD_REMOTE_TABLES);

  // Create a holding place for converted remote field values.
  // We need at least one valid record (with valid fields) in the
table.
```

```
  CAddressRecord remRecord(*m_RemRealTable, 0);

  if (m_RemRealTable->AppendBlankRecord(remRecord))
    return(CONDERR_BAD_REMOTE_TABLES);
  else if ((retval = AllocateRawRecordMemory(rawRecord,
ADDRESS_RAW_REC_MEM)))
    return(retval);

  // Set each PC record with statusFlag = None to Pending
  err = locIterator.FindFirst(locRecord, TRUE);
  while (!err)
  {
    if ((!locRecord.IsArchived()) && locRecord.IsNone())
      locRecord.SetStatus(fldStatusPENDING);

    err = locIterator.FindNext(locRecord, TRUE);
  }

  err = 0;
  rawRecIx = 0;
  while (!err && !retval)
  {
    rawRecord.m_RecIndex = rawRecIx;

    // Read && Convert each remote record from raw format to
    // CAddressRecord
    if (!(err = SyncReadRecordByIndex(rawRecord)))
    {
      // Convert from raw record format to CAddressRecord
      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;
    }
```

```
    rawRecIx++;
  }
  if (err != SYNCERR_FILE_NOT_FOUND)
      LogBadReadRecord(err);

  // Free the memory allocated for the raw record
  if (rawRecord.m_TotalBytes > 0 && rawRecord.m_pBytes)
    delete rawRecord.m_pBytes;

  // Send all modified records to the Palm OS device. Give a
  // specific record object.
  if (!retval)
  {
    CAddressRecord  loc2Rec(*m_LocRealTable, 0);
    retval = SendRemoteChanges(loc2Rec);
  }

  return(retval);
}
```

CopyRecordsPCtoHH

Prototype `long CopyRecordsPCtoHH (void)`

Parameters None

Purpose Copies all records from the PC to the device with the exception of records marked for archiving or deletion. Records marked for archiving are added to the archive table and later added to the appropriate archive files.

Example

```
long CAddressConduitMonitor::CopyRecordsPCtoHH(void)
{
  long             retval = 0, err = 0;
  CAddressRecord   locRecord(*m_LocRealTable, 0);
  CBaseIterator    locIterator(*m_LocRealTable);
```

```
  // Delete all Remote (Handheld) records
  if (SyncPurgeAllRecs(m_RemHandle))
  {
    retval = CONDERR_REMOTE_RECS_NOT_PURGED;
    return(retval);
  }

  // For each PC record ...
  err = locIterator.FindFirst(locRecord, FALSE);
  while (!err && !retval)
  {
    if (locRecord.IsArchived())
    {
      // Add PC record to Archive table
      retval = ClearStatusAddRecord(locRecord, *m_LocArchTable);
      // Mark for deletion
      locRecord.SetStatus(fldStatusDELETE);
    }
    else if (!locRecord.IsDeleted()) // record not deleted
    {
      // Add the record to the Handheld by virtual worker
      //function.
      locRecord.SetStatus(fldStatusNONE);
      if (retval = AddRemoteRecord(locRecord))
        LogBadAddRecord(locRecord);
    }
    err = locIterator.FindNext(locRecord, FALSE);
  }

  // Purge all deleted records from the PC table
  if (!retval)
    retval = m_LocRealTable->PurgeDeletedRecords();

  return(retval);
}
```

CopyRecordsHHtoPC

Prototype `long CopyRecordsHHtoPC (void)`

Parameters None

Purpose Copies all the records from the Palm OS device to the PC except for the records marked for archiving or deletion. Records marked for archiving are added to the archive table and later added to their appropriate archive files.

Example

```
long CAddressConduitMonitor::CopyRecordsHHtoPC(void)
{
  long            retval = 0, err = 0;
  CRawRecordInfo  rawRecord;
  WORD            recIx = 0;

  memset(&rawRecord, 0, sizeof(CRawRecordInfo));
  rawRecord.m_FileHandle = m_RemHandle;   // remote file handle
  rawRecord.m_RecIndex = recIx;

  if (!m_RemRealTable)
    return(CONDERR_BAD_REMOTE_TABLES);

  // Create a holding place for converted remote field values.
  // We need at least one valid record (with valid fields) in
  // the table.
  CAddressRecord remRecord(*m_RemRealTable, 0);
  if (m_RemRealTable->AppendBlankRecord(remRecord))
    return(CONDERR_BAD_REMOTE_TABLES);

  // Allocate memory for rawRecord.m_pBytes, return if Bad!
  if (retval = AllocateRawRecordMemory(rawRecord,
                                       ADDRESS_RAW_REC_MEM))
    return(retval);

  // Read in each Palm OS device record one at a time
  while (!retval && !err)
```

```
  {
    rawRecord.m_RecIndex = recIx ;
    if (!(err = SyncReadRecordByIndex(rawRecord)))
    {
      // Convert from raw record format to CAddressRecord
      if (!m_pDTConvert->ConvertFromRemote(remRecord, rawRecord))
      {
        if (remRecord.IsArchived())
          // Add device record to Archive table
          retval = ClearStatusAddRecord(remRecord,
                                              *m_LocArchTable);

        else if (remRecord.IsDeleted() == FALSE)
          // Add device record to PC table
          retval = ClearStatusAddRecord( remRecord,
                                              *m_LocRealTable);
      }
      else
        retval = CONDERR_CONVERT_FROM_REMOTE_REC;
    }
    recIx++;
  }
  if (err != SYNCERR_FILE_NOT_FOUND)
      LogBadReadRecord(err);

  // Free memory for rawRecord data
  if (rawRecord.m_TotalBytes > 0 && rawRecord.m_pBytes)
    delete rawRecord.m_pBytes;

  // Delete all records marked for deletion on the handheld
  if (!retval && SyncPurgeDeletedRecs(m_RemHandle))
    LogBadPurge(CONDERR_REMOTE_RECS_NOT_PURGED);

  return(retval);
}
```

LogRecordData

Prototype

```
long LogRecordData(  CBaseRecord& rRec,
                     char* fieldInfo)
```

Parameters

rRec	Pointer to a CBaseRecord.
fieldInfo	Buffer to store field values.

Purpose Adds information about a record to the log.

Return Codes None

Example

```
void CAddressConduitMonitor::LogRecordData(CBaseRecord& rRec,
                                            char * errBuff)
{
  CAddressRecord   &rLocRec = (CAddressRecord&)rRec;
  CString          csStr;
  int              len = 0;

  rLocRec.GetName(csStr);
  len = csStr.GetLength();
  if (len > 20)
    len = 20;

  strcpy(errBuff, "       ");
  strncat(errBuff, csStr, len);
  strcat(errBuff, ", ");

  rLocRec.GetFirst(csStr);
  len = csStr.GetLength();
  if (len > 20)
    len = 20;

  strncat(errBuff, csStr, len);
  strcat(errBuff, ", ");
```

```
  rLocRec.GetCompany(csStr);
  len = csStr.GetLength();
  if (len > 30)
    len = 30;

  strncat(errBuff, csStr, len);
}
```

LogApplicationName

Prototype `long LogApplicationName (char* appName, WORD len)`

Parameters

appName	Buffer in which to store the application name.
len	Length of `appName`.

Purpose Retrieves the application name, for example "Address Book" that will appear in the log.

Return Codes None

Example

```
void CAddressConduitMonitor::LogApplicationName(char* appName,
                                                 WORD len)
{
  // Load string from the resource file.
  ::LoadString(m_DllInstance, IDSTR_ADDRESSBOOK, appName, len);
}
```

CBaseMonitor Functions You May to Override

The following section provides a list of virtual functions that a subclass of `CBaseMonitor` may choose to override. Note that for these functions, the base class provides enough built-in functionality to allow any new conduits to execute and function without supplying code for these routines. However, the new conduit may want to disable some of the logic of the base

class by overriding some of these virtual functions and supplying essentially hollow code.

A `CBaseMonitor` subclass responsible for its own existing file formats will most likely override the following member functions:

- SaveLocalTables
- PurgeLocalDeletedRecs
- ApplyRemotePositionMap

Note that in contrast to the examples above, these examples can't be used as a template but just illustrate one way to implement the logic. The logic you need to implement may look completely different.

SaveLocalTables

Prototype

```
long SaveLocalTables (const char*)
```

Parameters

char* Disk file name to save records into.

Purpose

Writes all records residing in the data member `m_LocRealTable` to a disk file using the passed-in string as its name. By default, the base class commits the contents of the `m_LocRealTable` object to disk using MFC serialization logic.

Note that this function does have behavior if you don't override it. If you want data formats that differ at all from those in the four native device application, you must override it.

If your conduit synchronizes with a file format different from the MFC serialization provided by the `CBaseTable` class, it should override this member function. The core synchronization logic invokes this routine after all data has been exchanged with the device. The data present in the `m_LocRealTable` data member are fully synchronized at the time this function is called. This would be the logical point to convert the records contained in the `m_LocRealTable` object into the format required as output from the conduit.

Return Codes

0 = success

-1 = could not save the data

Default Logic

```
// This is the default logic in the base class
long CBaseConduitMonitor::SaveLocalTables (const char* fileName)
{
     CString destFile(fileName);

     long retval = m_LocRealTable->SaveTo(destFile);
     return(retval);
}


//
// This shows how a Monitor subclass may override the member
// function and invoke its own processing logic for the
// freshly synchronized table object. The new data member
// m_Generator is assumed to have been created during the
// overridden version of the ObtainLocalTables() member function.
//
long CMyConduitMonitor::SaveLocalTables(const char* fileName)
{
     long lErr = 0;
     CString destFile(fileName);

    if (m_Generator) {
        CAddressRecord addrRec(*m_LocRealTable);
        lErr = m_Generator->PostProcessTables
                                (*m_LocRealTable, addrRec);
    }
    return(lErr);
}
```

PurgeLocalDeletedRecs

Prototype `long PurgeLocalDeletedRecs (void)`

Parameters None

Purpose The default logic (provided by the base class `CBaseMonitor`) of this routine iterates through the data member `m_LocRealTable` and physically removes each record marked for deletion. The native synchronization logic processing or the desktop software may have marked records for deletion. It's not necessary to override this function unless a conduit's concerned about proprietary file formats.

If your conduit synchronizes with a file format that's different from the MFC serialization provided by `CBaseTable`, it should override this member function. The core synchronization logic invokes this routine before `SaveLocalTables`. A conduit performing a post-processing pass on the `m_LocRealTable` object may actually want deleted records to remain in the table so it can detect them. For this to occur, the conduit may need to override this member function just to make it inactive.

Return Codes 0 = success

-1 = could not purge the data

Default Logic

```
// This is the default logic in the base class
long CBaseConduitMonitor::PurgeLocalDeletedRecs()
{
     long retval = m_LocRealTable->PurgeDeletedRecords();
     return(retval);
}

//
// This shows how a Monitor subclass may override the member
// function and supply no code, in effect neutralizes this
// function.

long CMyConduitMonitor::PurgeLocalDeletedRecs()
{
     return(0);
}
```

ApplyRemotePositionMap

Prototype `long ApplyRemotePositionMap (void)`

Parameters None

Purpose The default logic (provided by the base class `CBaseMonitor`) issues a request to the device asking for a sorted list of its record IDs. Once obtained, it's applied to the order of records in the `m_LocRealTable` table object. As a result, the desktop software will display its records in the same order as the device.

If the destination for the synchronized data is a proprietary file format, your conduit needs to override this function so it does nothing. This saves execution time by eliminating unnecessary traffic over the serial link.

Return Codes 0 = success

-1 = could not apply the cross mapping

Default Logic

```
// This is the default logic in the base class
long CBaseConduitMonitor::ApplyRemotePositionMap()
{
     // To view this code look in the basemon.cpp file
     // residing in the \condsdk\src directory.
}


//
// This is shows how a Monitor subclass may override the member
// function and supply no code, which in effect neutralizes this
// function.

long CMyConduitMonitor::ApplyRemotePositionMap()
{
     return(0);
}
```

Creating a CBaseDTLinkConverter Subclass

Any conduit has to convert record layouts between those on the device and those on the PC. If your conduit exchanges data with the native applications on the device, you can use the one of the subclasses of `CBaseDTLinkConverter` which are provided as sample code. Otherwise, you must create a subclass and initialize data members and provide virtual functions as necessary.

The Conduit.DLL is responsible for adhering to the proper data structures. Conduits pass records destined for the device through a link converter, which formats records in a layout to match the device record layout. This conversion facilitates data storage on the device. The conduits provided with the Desktop pass any raw record data retrieved from the device through their own link converter, which formats the data into a layout that matches the record layout on the PC. Using this link converter streamlines the development process by utilizing the existing record synchronization logic used between the device and the Desktop to facilitate record comparisons during the synchronization process.

The `CBaseDTLinkConverter` instance is created by an instance of `CBaseMonitor` or by an instance of one of its subclasses and stored inside that instance. The converter understands the remote database record layouts and converts them into a form the native synchronization logic can use.

You learn about these aspects of a `CBaseDTLinkConverter`:

- CBaseDTLinkConverter Basic Structure
- CBaseDTLinkConverter Data Members
- CBaseDTLinkConverter Functions You Must Override
- CBaseDTLinkConverter Functions You May Override

CBaseDTLinkConverter Basic Structure

This section provides a brief introduction to the most important aspects of the link converter:

- The Log Object
- Casting of Member Functions
- Carriage Returns and Line Feeds

The Log Object

The `CBaseMonitor` gives the `CBaseDTLinkConverter` a `CSyncLog` data member. This log object allows the link converter to record important events that the desktop software user can view. The `CSyncLog` class is defined in TABLES.DLL

Casting of Member Functions

Just as for `CBaseMonitor`, many of the member function prototypes define a parameter as a `CBaseRecord&`, then rely on the code you provide to cast the parameter to the specific record object (an instance of a subclass of `CBaseRecord`) that the conduit is synchronizing. This makes it possible to have all conduits use the same core logic.

Carriage Returns and Line Feeds

All converters must deal with the carriage return/ line feeds issue. The Palm OS device uses Macintosh-style text conventions; it allows only line feeds but not carriage returns embedded in any of its text fields. Conversely, in the PC/conduit environment, carriage returns appear in text fields along with new lines. As a result, the converter has to do the following:

- **From PC to Device**. A converter must strip all carriage returns from the text fields of a given record before sending them to the Palm OS device. If a converter fails to strip out carriage returns, the device applications may not be able to handle the new data.
- **From Device to PC.** A converter must add carriage returns into all text fields (which contain only new lines) coming from the Palm OS device.

CBaseDTLinkConverter Data Members

The `CBaseDTLinkConverter` class contains a few data members which assist in performing the data conversion. These data members are maintained by the `CBaseDTLinkConverter` class and made available for use by subclasses.

Normally the code for a converter is placed in a separate source file from that of the monitor. The four conduits provided with the Palm OS Desktop software each have a source file which holds the converter code (ADDLINK.CPP, TODLINK.CPP, DATLINK.CPP, or MEMLINK.CPP). Listing 5.8 is an excerpt of the class definition for the

`CBaseDTLinkConverter` class present in the header file BASEMON.H from the conduit SDK.

Listing 5.8 Base Converter Data Members

```
//
// Base Converter class
//
class CBaseDTLinkConverter {
protected:
     CSyncLog*   m_pLog;
     TCHAR*      m_TransBuff;
     HINSTANCE   m_DllInstance;
};
```

CSyncLog* m_pLog

A pointer to the log object created by the HotSync program and handed into the link converter class as a parameter on its constructor line. The log is available for recording short statements that alert the end user about actions to take. The link converter should neither create nor destroy this pointer.

TCHAR* m_TransBuff

A pointer to memory which is allocated/destroyed by the `CBaseDTLinkConverter` class. No subclass should attempt to maintain this memory pointer. This memory buffer is used by some of the inherited utility functions which adds or removes line feeds aon string buffers that are exchanged with the device.

HINSTANCE m_DllInstance

This instance handle is passed in on the constructor line, and originates from the `OpenConduit` startup routine. This instance handle is made available to the converter should it decide to extract strings from a resource file for use in a log entry. It can also be used for other Windows-related functions which need an instance handle.

CBaseDTLinkConverter Functions You Must Override

This section first provides a list and then the definition and purpose of the constructor and virtual functions that a class derived from `CBaseDTLinkConverter` is required to override. By default, the base class version of the virtual functions do nothing. Unless the subclass supplies working code for them, the conduit will fail to convert any records from the Palm OS device.

- CAddressDTLinkConverter Constructor and Destructor
- ConvertToRemote
- ConvertFromRemote
- ConvertToRemoteCategories
- ConvertFromRemoteCategories

CAddressDTLinkConverter Constructor and Destructor

Like every C++ class, the link converted needs a constructor and destructor. The constructor is discussed in some detail, the destructor is a standard C++ destructor.

Prototype

```
CAddressDTLinkConverter (  CSyncLog* pLog,
                           HINSTANCE hInst)
```

Parameters

pLog	Pointer to a log object (may be NULL).
hInst	Instance handle of the DLL.

Purpose

Each class derived from `CBaseDTLinkConverter` must supply its own constructor. It's important that this constructor pass both parameters to its base class constructor, where they are stored on the data members that are then inherited to the subclass.

Return Codes

None

Example

```
CAddressDTLinkConverter::CAddressDTLinkConverter
                  (CSyncLog* pLog, HINSTANCE hInst)
                  :CBaseDTLinkConverter(pLog, hInst)
{
}
```

ConvertToRemote

Prototype

```
long ConvertToRemote (CBaseRecord &rRec,
                      CRawRecordInfo &rInfo)
```

Parameters

rRec	Reference to a PC format record object to supply unconverted information.
rInfo	Reference to a structure containing a buffer for device format record data.

Purpose

This member function prepares a data record for transmission from the PC to the device. The first parameter contains a valid PC database record (in the form of a `CBaseRecord` subclass) whose data must be extracted and formatted into the a format the Palm OS device can read. The second parameter has a memory buffer `rInfo.m_pBytes` large enough to contain the device version of the record data. The function must place the newly formatted record information into this memory buffer, which the SyncManager transmits over to the device.

The example illustrates how to populate the `CRawRecordInfo` structure handling both the fixed portion and variable length portion of an Address Book record. Also shown is how the PC has to strip any carriage returns out of its text fields before sending them to the device. This is a required activity and if omitted, may cause the device to crash when its application attempts to read that data.

Return Codes

0 = success

-1 = failed to convert record

Example Body

```
CAddressDTLinkConverter::ConvertToRemote(CBaseRecord& rRec,
                                         CRawRecordInfo& rInfo)
{
     long retval = 0;

     CAddressRecord& rAddrRec = (CAddressRecord &)rRec;
                                     // cast to proper class
     rInfo.m_RecSize = 0;
```

```
rAddrRec.GetRecordId(tempInt);
                                    // set RecordID
rInfo.m_RecId = (long)tempInt;
rAddrRec.GetCategoryId(tempInt);
                                    // set Category ID
rInfo.m_CatId = tempInt;
rInfo.m_Attribs = 0;

if (rAddrRec.IsPrivate())
                                    // deal with attributes
     rInfo.m_Attribs |= PRIVATE_BIT;
if (rAddrRec.IsArchived())
     rInfo.m_Attribs |= ARCHIVE_BIT;
if (rAddrRec.IsDeleted())
     rInfo.m_Attribs |= DELETE_BIT;
if (rAddrRec.IsModified() || rAddrRec.IsAdded())
     rInfo.m_Attribs |= DIRTY_BIT;
pBuff = (char*)rInfo.m_pBytes;
                                    // get a handy pointer

// Last Name field
retval = rAddrRec.GetName(tempStr);
len = tempStr.GetLength();
if (len != 0) {
     flags.name = 1;
     // Strip the CR's (if present)
     //place result directly into pBuff
     pSrc    = tempStr.GetBuffer(len);
     destLen = StripCRs(pBuff, pSrc, len);
     tempStr.ReleaseBuffer(-1);
     pBuff += destLen;
     rInfo.m_RecSize += destLen;
       // accumulate variable length
}

// FirstName field
retval = rAddrRec.GetFirst(tempStr);
len = tempStr.GetLength();
```

```
    if (len != 0) {
         flags.firstName = 1;
         // Strip the CR's (if present)
         // place result directly into pBuff
         pSrc    = tempStr.GetBuffer(len);
         destLen = StripCRs(pBuff, pSrc, len);
         tempStr.ReleaseBuffer(-1);
         pBuff += destLen;
         rInfo.m_RecSize += destLen;
           // accumulate variable length
    }

           // Deal with all other fields...
    return(retval);
}
```

ConvertFromRemote

Prototype

```
long ConvertFromRemote ( CBaseRecord &rRec,
                         CRawRecordInfo &rInfo)
```

Parameters

rRec Reference to PC format record object to receive converted information.

rInfo Reference to structure holding a record from device.

Purpose

Converts the remote data record (which exists in the second parameter `rInfo` in as packed bytes), into data that can be set into the first parameter (a `CBaseRecord` subclass). Information is pulled out of the `rInfo` parameter and set into the `rRec` parameter.

Subclasses must override this routine; the `CBaseRecord` instance arriving in the first parameter must be cast to the specific record object used on the local PC database. Shown here is an example of adding carriage returns into text fields (Last Name), coming from the Palm OS device that contain only new lines.

Return Codes

0 = success

-1 = failed to convert record

Example

```
CAddressDTLinkConverter::ConvertFromRemote (CBaseRecord&  rRec,
                                            CRawRecordInfo& rInfo)
{
     long retval = 0;

     CAddressRecord& rAddrRec = (CAddressRecord &)rRec;
     rAddrRec.SetRecordId(rInfo.m_RecId);
                              // grab and set the record Id
     rAddrRec.SetCategoryId(rInfo.m_CatId);
                              // grab and set Category Id

     if (rInfo.m_Attribs & ARCHIVE_BIT)
                              // check and set archive flag
          rAddrRec.SetArchiveBit(TRUE);
     else
          rAddrRec.SetArchiveBit(FALSE);

     if (rInfo.m_Attribs & PRIVATE_BIT)
                              // check and set private flag
          retval = rAddrRec.SetPrivate(TRUE);
     else
          retval = rAddrRec.SetPrivate(FALSE);

     retval = rAddrRec.SetStatus(fldStatusNONE);
                              // clear record status field

     if (rInfo.m_Attribs & DELETE_BIT)
                              // check and set Delete status
          retval = rAddrRec.SetStatus(fldStatusDELETE);
     else if (rInfo.m_Attribs & DIRTY_BIT)
                              // check and set Modified status
          retval = rAddrRec.SetStatus(fldStatusUPDATE);

     // Only convert body if remote record is *not* deleted..

     if (!(rInfo.m_Attribs & DELETE_BIT))  {
```

```
        pBuff = (char*)rInfo.m_pBytes;
                                    // get a handy pointer

        // Last Name field (deal with adding carriage returns)
        if (flags.name) {
             // Add any necessary CRs,
             //result is placed in m_TransBuff
             AddCRs(pBuff, strlen(pBuff));
             aString = m_TransBuff;
             retval = rAddrRec.SetName(aString);
             pBuff += strlen(pBuff) + 1;
        }
        else
             retval = rAddrRec.SetName(csEmpty);
        // FirstName field (deal with adding carriage returns)
        if (flags.firstName) {
             // Add any necessary CRs,
             //result is placed in m_TransBuff
             AddCRs(pBuff, strlen(pBuff));
             aString = m_TransBuff;
             retval = rAddrRec.SetFirst(aString);
             pBuff += strlen(pBuff) + 1;
        }
        else
             retval = rAddrRec.SetFirst(csEmpty);

        // Convert all other fields.......

    }
    return(retval);
}
```

ConvertToRemoteCategories

Prototype

```
long ConvertToRemoteCategories(
                         CDbGenInfo& dbInfo,
                         CCategoryMgr* catMgr)
```

Parameters

dbInfo	Reference to an object containing the `AppInfoBlock`.
pCatMgr	Pointer to a CategoryManager object.

Purpose

Prepares the `AppInfoBlock` structure (which contains the categories and is contained in `dBInfo`) to be sent to the Palm OS device. The second parameter holds the synchronized categories (in a PC formatted object) which need to be converted and placed into the first parameter.

Each Palm OS database stores its categories inside the `AppInfoBlock` along with other proprietary information. The categories exist at a well-known byte offset into this AppInfoBlock.

The utility routine `ReplaceCategories`, which moves categories from the `CategoryManager` object to the `AppInfoBlock`, is defined in the `CBaseDTLinkConverter` class that all its subclasses may invoke.

Return Codes

0 = success

CONDERR_CONVERT_TO_REMOTE_CATS

Example

```
CAddressDTLinkConverter::ConvertToRemoteCategories
                                        (CDbGenInfo&   dbInfo,
                                        CCategoryMgr* catMgr)
{
     long retval = CONDERR_CONVERT_TO_REMOTE_CATS;
     char* pBuff;

     if (dbInfo.m_pBytes) {
          pBuff = (char*)dbInfo.m_pBytes;
          *((WORD *)pBuff) = 0;
                              // Clear the category dirty flags
          pBuff += sizeof(WORD);
                              // offset to specific spot for cats
          retval = CBaseDTLinkConverter::ReplaceCategories
                                                  (pBuff, catMgr);
    }
     return(retval);
}
```

ConvertFromRemoteCategories

Prototype

```
long ConvertFromRemoteCategories
         (CDbGenInfo& dbInfo, CCategoryMgr* catMgr)
```

Parameters

dbInfo	Reference to an object containing the `AppInfoBlock`.
pCatMgr	Pointer to a CategoryManager object.

Purpose

Extracts the category strings and IDs (that have just been delivered from the device) from `dbInfo` and places them into `catMgr`.

Categories generally exist at a well-known byte offset into the `AppInfoBlock`, and a given subclass should know its particular placement of categories. A utility routine defined in the `CBaseDTLinkConverter` is available for all subclasses to assist in extracting raw category information to place into the PC-formatted CategoryManager object.

Return Codes

0 = success

CONDERR_CONVERT_TO_LOCAL_CATS

Example

```
CAddressDTLinkConverter::ConvertFromRemoteCategories
(CDbGenInfo&  dbInfo,

CCategoryMgr* catMgr)
{
      long retval = CONDERR_CONVERT_TO_LOCAL_CATS;
      char* pBuff;
      short wTemp;

      if (dbInfo.m_pBytes) {
           pBuff = (char*)dbInfo.m_pBytes;
           wTemp = *((WORD*)pBuff);
           wTemp = FlipWord(wTemp);
                        // two byte words arrive in Motorola format

           pBuff += sizeof(WORD);
                        // offset into category area
```

```
        retval = CBaseDTLinkConverter::ExtractCategories
                                          (pBuff, wTemp, catMgr);
    }
    return(retval);
}
```

CBaseDTLinkConverter Functions You May Override

A class derived from `CBaseDTLinkConverter` can optionally override these virtual member functions.

```
virtual long ConvertPositionMap(
                             CPositionInfo &rInfo);
```

The routine `ConvertPositionMap` does perform some processing by default.

It runs through the record ID's and flips the DWords to Intel format.

```
virtual void SynchronizeAppInfoBlock(
                             CDbGenInfo& dbInfo,
                             CBaseTable& rTable,
                             eSyncTypes syncType,
                             eFirstSync firstSync)
```

This routine does nothing but is available in case a conduit is aware of custom information stored in the `AppInfoBlock` by an application residing on the device; in effect any information except the categories, which are handled separately.

CBaseDTLinkConverter Utility Member Functions

The following member functions are available for all subclasses of `CBaseDTLinkConverter`. They help you deal with date formats arriving from the Palm OS device. They also help you deal with two and four byte integer values that exist in Motorola format on the device and must be flipped to Intel format on the PC. Other utility functions to assist in strip-

ping and adding carriage returns into text fields, and extracting category strings and IDs.

```
long ConvertToTdDate(  TdDateType& rTdDate,
                      TdTimeTyp& rTdTime,
                        long lDate);
```

```
long ConvertFromTdDate ( TdDateType& rTdDate,
                        TdTimeType& rTdTime,
                        long& rDate);
```

```
unsigned long SwapDWordToMotor(unsigned long);
```

```
unsigned long SwapDWordToIntel (unsigned long);
```

```
unsigned short FlipWord (unsigned short);
```

```
int StripCRs (TCHAR* pDest, TCHAR* Src, int len);
```

```
long AddCRs (TCHAR* pSrc, int len);
```

```
long ExtractCategories(  char *catLabelsPtr,
                        short dirtyCats,
                        CCategoryMgr* catMgr);
```

```
long ReplaceCategories(  char *catLabelsPtr,
```

```
                    CCategoryMgr* catMgr);
```

Creating a CBaseTable Subclass

The information the conduit uses to synchronize an application on the device with one on the PC is stored in two table objects, which must both be instances of `CBaseTable` or one of its subclasses. The tables are used as follows:

- The conduit loads all locally stored records of an application in an instance that's the `m_LocRealTable` data member of `CBaseMonitor`.
- The conduit then retrieves remote records, one at a time and stores them in the instance that's the `m_RemRealTable` data member of `CBaseMonitor`. It compares each record with the record that has the same recordID (record IDs are assigned by the device).
- Most applications also provide a backup table that is used during SlowSync operations.
- If an application allows users to archive records that they deleted on the device, it also has to provide an archive table. The four native application allow users to do this.

If you have decided to use the native synchronization logic, the work you must do with the tables is actually rather limited. However, because it's useful you both understand what you must do and why you must do it, this section actually discusses this topic from both points of view:

- How to Set Up Tables provides step by step instructions for setting up the tables.
- More About Tables provides more detailed information about what the tables do, including some code examples from the ToDo application.

How to Set Up Tables

This section explains what an application needs to do to synchronize records and categories appropriately using the native synchronization logic.

The process differs slightly depending on the application; see the source code of the four native applications for examples of similar but different setups. The examples in this section come mostly from the ToDo conduit

because its records are more complex than those of the address book used in previous sections.

To use the native synchronization logic, you need to do the following:

1. Create a subclass of `CBaseTable`. This class is the "glue" that holds all things together; some of the information it needs is in the classes associated with it (which you create in the steps below).
   - Create a subclass of `CBaseTable` with an appropriate constructor and destructor.
   - Override the virtual function `AppendDuplicateRecord`. The function lets each record work on all its fields; it takes care of the details of copying from one record to a new record. See AppendDuplicateRecord function from ToDo base table.
   - If your application requires specialized sorting, optionally override `AppendBlankRecord`.

   Note that you don't have to override the standard `OpenFrom` and `SaveTo` functions; the functions use the information in your subclass of `CBaseSchema` to determine how to write the data in and out.
2. Create a subclass of `CBaseSchema` with an appropriate constructor and destructor and override the `DiscoverSchema` virtual function. (see DiscoverSchema function from CToDoSchema)

   The schema is a template of the record, the table uses that information when synchronizing the record.
3. Create a subclass of `CBaseRecord`

   This subclass needs to have one virtual function for each application-specific field of the record. For example, for records in the ToDo PIM, functions `SetDescription`, `SetDueDate`, `SetCompleted`, `SetPriority`, and so on are provided. The record inherits the fields Status, RecordID and Category ID, so your subclass does not need to take care of them (see DiscoverSchema function from CToDoSchema)

   Note that if the record class and the schema class don't agree on the fields in your records, problems will result.
4. Create a subclass of `CBaseIterator`

   The iterator class contains behavior for sorting and finding things; functions that apply to all records at once, for example, sorting by field. You must override its virtual functions with functions that call the same function in the base class. You may also decide to add

functions to your class that perform special actions, for example, sorting by priority.

More About Tables

This section provides more detailed information about the classes that allow `CBaseMonitor` to synchronize your database and its records. For each class, you learn about the functions you must override, likely or required additions, and some information about the inherited behavior as appropriate. Code examples from the ToDo conduit are included.

You learn about these classes:

- CBaseTable Class
- CBaseIterator Class
- CBaseSchema Class
- CBaseRecord Class

Note that all table classes can take advantage of a lot of prepackaged functionality provided in bfields.h. This includes the ability to sign things, different operators, and functionality that helps with serializing fields. For example, if you tell a field to serialize itself, it knows how to do it. When you define something as an integer field, you get a lot of functionality with it.

CBaseTable Class

A conduit using the native synchronization logic uses four table instances while it's executing: local table, remote table, archive table, and backup table (SlowSync only). Each table has to be an instance of the same subclass of `CBaseTable`.

From `CBaseTable`, the table inherits some behavior as well as places to store pointers to the schema, record, and iterator objects. These objects contain some of the application-specific record information and are discussed below.

`CBaseTable` is defined in basetable.h

The only virtual function you must override in `CBaseTable` is `AppendDuplicateRecord`. Here's an example from the ToDo conduit.

Listing 5.9 AppendDuplicateRecord function from ToDo base table.

```
///////////////////////////////////////////////////////////////////
// Function:     AppendDuplicateRecord()
//
// Description:  Appends a new blank record then fills it with
//               the passed parameter 'rFromRec'.
//
//               Allows a new set of fields (a row) to be added
//               to table object. The set of fields comprises one
//               full record, and initially each has blank data
//               Next the passed in record object is used as a
//               source of fields whose values are duplicated in
//               the newly appended 'blank' set of fields.
//
//   *Note*        Generally only called from the ConduitMonitor /
/                 object during synchronization procedures.
//
// Parameters:
// rFrom    - Record object to copy data from
// rTo      - Ends up positioned at the new row of fields
                   in the table
// bAllFlds - If true replicates ALL fields including **RecordID*
//          - If false does *not* duplicate the recordId or Status
//
// Returns:  0 - Success
///////////////////////////////////////////////////////////////////
long CToDoTable::AppendDuplicateRecord(CBaseRecord& rFrom,
                                       CBaseRecord& rTo, BOOL
bAllFlds)
{
  int     tempInt;
  CString tempStr;
  long    tempLong, len, retval = -1;

  CToDoRecord& rFromRec = (CToDoRecord&)rFrom;
  CToDoRecord& rToRec = (CToDoRecord&)rTo;
```

```
//
// Source record must be positioned at valid data.
//
if (rFromRec.m_Positioned)
{
  if (!CBaseTable::AppendBlankRecord(rToRec))
  {
    if (bAllFlds)
    {
      if (!rFromRec.GetRecordId(tempInt))
        retval = rToRec.SetRecordId(tempInt);

      if (!(retval = rFromRec.GetStatus(tempInt)))
        retval = rToRec.SetStatus(tempInt);

      retval = rToRec.SetArchiveBit(rFromRec.IsArchived());
    }
    if (!retval && !rFromRec.GetDescription(tempStr))
      retval = rToRec.SetDescription (tempStr);

    if (!retval && !rFromRec.GetDueDate(tempLong))
      retval = rToRec.SetDueDate(tempLong);

    if (!retval)
      retval = rToRec.SetCompleted(rFromRec.IsCompleted());

    if (!retval && !rFromRec.GetPriority(tempInt))
      retval = rToRec.SetPriority(tempInt);

    if (!retval)
      retval = rToRec.SetPrivate(rFromRec.IsPrivate());

    if (!retval && !rFromRec.GetCategoryId(tempInt))
      retval = rToRec.SetCategoryId(tempInt);

    if (!retval)
    {
      rFromRec.GetNote(tempStr);
```

```
        len = tempStr.GetLength();
        if (len > 0)
          retval = rToRec.SetNote(tempStr);
      }
    }
  }
  return(retval);
}
```

The table class relies on a schema, record, and iterator object for information about the records your conduit synchronizes. You therefore must create subclasses of `CBaseSchema`, `CBaseRecord`, and `CBaseIterator`, discussed in the next three sections.

CBaseRecord Class

The CBaseRecord class is one of the places where information about your records is stored.

This information is actually made known to the system in several ways:

- **FieldIDs** provide the ID for each field in your conduit's header file. Here's a partial example from tdtable.h, which defines the To Do table class:

  ```
  #define tdFLDRecordID    0
  #define tdFLDStatus      1
  #define tdFLDPosition    2
  #define tdFLDDesc        3
  #define tdFLDDueDate     4
  ```

- A `DiscoverSchema` function you must supply inside your schema subclass that defines the template of the record (see DiscoverSchema function from CToDoSchema).
- A virtual set and a virtual get function for each record, for example, `SetDescription` and `GetDescription` or `SetDueDate` and `GetDueDate`. Each function fills in the corresponding record using the information in the schema.
  The MODFILTER_STUPID flag set by this function sets the record dirty whenever it's touched. This is usually recommended.

Here's an example of the `SetDescription` function from the ToDo record.

Listing 5.10 SetDescription function in CToDoRecord

```
long CToDoRecord::SetDescription(CString& rDesc)
{
  BOOL autoFlip   = FALSE;
  int  currStatus = 0;
  long retval     = DERR_RECORD_NOT_POSITIONED;

  CStringField* pFld = NULL;

  if (m_Positioned && (pFld = (CStringField*)
m_Fields.GetAt(tdFLDDesc)))
  {
    if (m_wModAction == MODFILTER_STUPID)
    {
      GetStatus(currStatus);
      if (currStatus != fldStatusADD)
      {
        CStringField tmpFld(rDesc);
        if (pFld->Compare(&tmpFld))
          autoFlip = TRUE;
      }
    }
    if (!pFld->SetValue(rDesc))     // Set new field value
    {
      if (autoFlip)
        SetStatus(fldStatusUPDATE);
      retval = 0;
    }
  }
  return(retval);
}
```

CBaseSchema Class

The Schema class contains information the record object uses inside the SetDescription function to set up the record. Here's an example from the ToDo conduit:

Listing 5.11 DiscoverSchema function from CToDoSchema

```
long CToDoSchema::DiscoverSchema(void)
{
  m_FieldsPerRow = 10;
  m_FieldTypes.SetSize(m_FieldsPerRow);
  m_FieldTypes.SetAt(tdFLDRecordID,   (WORD)eInteger);
  m_FieldTypes.SetAt(tdFLDStatus,     (WORD)eInteger);
  m_FieldTypes.SetAt(tdFLDPosition,   (WORD)eInteger);
  m_FieldTypes.SetAt(tdFLDDesc,       (WORD)eString);
  m_FieldTypes.SetAt(tdFLDDueDate,    (WORD)eDate);
  m_FieldTypes.SetAt(tdFLDCompleted,  (WORD)eBool);
  m_FieldTypes.SetAt(tdFLDPriority,   (WORD)eInteger);
  m_FieldTypes.SetAt(tdFLDPrivate,    (WORD)eBool);
  m_FieldTypes.SetAt(tdFLDCategoryID, (WORD)eInteger);
  m_FieldTypes.SetAt(tdFLDNote,       (WORD)eString);

  // Be sure to set the 3 common fields' position
  m_RecordIdPos     = tdFLDRecordID;
  m_RecordStatusPos = tdFLDStatus;
  m_CategoryIdPos   = tdFLDCategoryID;
  m_PlacementPos    = tdFLDPosition;

  return(0);
}
```

CBaseIterator Class

The `CBaseIterator` class holds functions that perform actions on all records, such as searching and sorting them. Here's how the class is defined at the top level:

Listing 5.12 CBaseIterator Class

```
class TABLES_DECL CBaseIterator
{
public:

       CBaseIterator         (CBaseTable&);
      ~CBaseIterator         ();

  long UnSort                (void);
  long SortByRecordId        (void);
  long SortByRecordStatus    (void);
  long SortByCatId           (void);
  long SortByPlacementField (void);
  long FindFirst             (CBaseRecord&,
                                  BOOL skipDels = TRUE);
  long FindNext              (CBaseRecord&,
                                  BOOL skipDels = TRUE);
  long FindByRecordId        (int nRecId, CBaseRecord&,
                                  BOOL skipDels = TRUE);
  long FindByCatId           (int nCatId, CBaseRecord&,
                                  BOOL skipDels = TRUE);
  long FindByPlacementField (int nPlaceKey, CBaseRecord&,
                                  BOOL skipDels = TRUE);

  long GetAt                 (CBaseRecord&, long lRowOffset);
  long GetCurrentRowPosition(long& rCurrRowOffset);

//long  SetTableDirty  (void);      // Flips m_ValidTable to FALSE
};
```

Considering Category Manager Modifications

Categories are “buckets” in the database to which records are assigned; they include, for example, Business and Personal or other, user-defined categories. The native applications always synchronize categories first; this is

done by BaseMonitor standard logic inside FastSync. After categories are set up, records are synchronized.

NOTE: Many developers find they can use the native category behavior as is.

There are several restrictions on using categories:

- The maximum number of categories is 16 (Palm OS device and desktop combined). This includes the category unfiled.
- By default, each category has an index, an ID, and a name.
- Category IDs are assigned on the device

The category manager is actually part of the tables library. The category manager knows how to add, delete, and rename categories. It knows about the categories' Index, ID, Name, and FileName.

In effect, the category manager knows how to find categories and serialize them in and out.

Listing 5.13 CategoryManager Class

```
class TABLES_DECL CCategoryMgr : public CObject
{
  DECLARE_SERIAL(CCategoryMgr)

public:

        CCategoryMgr();
        ~CCategoryMgr();

  void      DeleteAllCategories(BOOL bNailUnfiled = TRUE);

  CatError  GenFileName (CString& csFileName, CString& csCatName);
  int       GetFreeIndex ();
  int       GetNextAddID ();

  CatError Add(CCategory* pCategory);
  CatError Delete(int nIndex);
  CatError Rename(int nIndex, CString& csName);
```

```
  CatError FindFirst(CCategory*& pCategory);
  CatError FindNext(CCategory*& pCategory);
  CatError FindName(CString& csName, CCategory*& pCategory);
  CatError FindIndex(int nIndex, CCategory*& pCategory);
  CatError FindID(int nID, CCategory*& pCategory);

  int GetCount()
    { return (m_Categories.IsEmpty() ? 0 :
m_Categories.GetCount()); }

  virtual void Serialize( CArchive& archive );
};
```

In most cases, no modifications to category behavior are required. If you do decide you need specialized category management, you have several options:

- Use the link converter to have categories correspond to the format expected by the native synchronization logic.
- Ignore categories altogether.
- Create a subclass of the CategoryManager class an provide virtual functions or data members to work with your categories.

6

SyncManager Function Calls

This chapter lists all SyncManager function calls, organized as follows:

- Session-Oriented Calls
- File-Oriented Calls
- Record-Oriented Calls
- Utility Calls

A complete list of error codes is provided in Error Codes.

Session-Oriented Calls

The session-oriented API consists of two calls.

- SyncRegisterConduit
- SyncUnRegisterConduit

SyncRegisterConduit

Purpose Check whether a conduit is registered. If it isn't, the conduit is registered internally by the HotSync manager.

Prototype

```
long SyncRegisterConduit (CONDHANDLE &)
```

Parameters

CONDHANDLE	Reference to CONDHANDLE that is populated.

Result SYNCERR_NONE
SYNCERR_COMM_NOT_INIT
SYNCERR_REMOTE_CANCEL_SYNC

Description — This routine is called when a conduit DLL first begins its synchronization activities. It has to be called by every conduit to prepare the device for synchronization. If the conduit doesn't make this call, synchronization cannot take place.

SyncUnRegisterConduit

Purpose — Unregister a conduit.

Prototype — `long SyncUnRegisterConduit (CONDHANDLE)`

Parameters

CONDHANDLE	Conduit handle received from a `SyncRegisterConduit` call.

Result — SYNCERR_NONE
SYNCERR_COMM_NOT_INIT

Description — This call allows the device to clean up memory and resources following synchronization.

File-Oriented Calls

The file-oriented function calls provide file manipulation of the databases on the device.

All remote databases exist on a memory card. In the first Pilot release, only one memory card is present on the device, referred to as card #0. A memory card may store databases in one of two areas, either RAM or ROM. When opening or creating a remote database, it is necessary to indicate which of the memory cards the database is to reside upon.

The figure below illustrates the layout of a remote Pilot database. It is not necessary to know this layout, however it does show the components that can be manipulated by the file-oriented API.

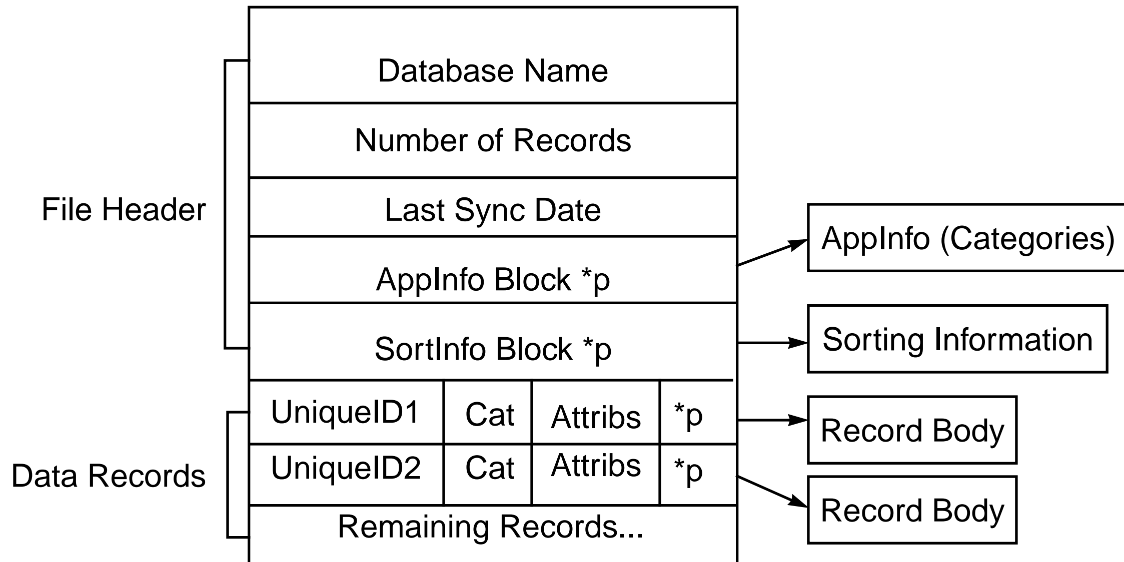


Figure 6.1 Remote Database Layout

The `AppInfoBlock` and `SortInfoBlock` are variable-length blocks of information that a caller can read/write to the database. For the four built-in Pilot applications, the AppInfoBlock contains the list of 16 category strings which are associated with a database. Currently the SortInfoBlock is unused by the built-in applications, however if a new database is created on the device, the caller may store whatever they wish in this variable length block. The figure also illustrates the fixed portion of data records containing the unique record ID (assigned by the Pilot operating system), the category ID, and an attributes byte, signifying the status of the individual record (Add/Modify/Delete). Records that are deleted through the API SyncDeleteRecord(), are actually only marked for deletion, where the attribute byte has a single bit set to indicate the record is deleted. To actually remove the physical space from the database which the (deleted) record occupies, the API SyncPurgeDeletedRecs() must be called.

The file-oriented API includes these calls:

- SyncCloseDB
- SyncCreateDB
- SyncDeleteDB
- SyncOpenDB

- SyncReadDBAppInfoBlock
- SyncReadDBSortInfoBlock
- SyncResetSyncFlags
- SyncWriteDBAppInfoBlock
- SyncWriteDBSortInfoBlock

SyncCloseDB

Purpose: Close the currently open database on the device

Prototype: `long SyncCloseDB (BYTE fHandle)`

Parameters: fHandle — Database file handle from an open or create call.

Result:
SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN

SyncCreateDB

Purpose: Create a new database on the Palm OS device.

Prototype: `long SyncCreateDB (CDbCreateDB& rDbStats)`

Parameters: rDbStats — Reference to a `CDbCreateDB` structure (see Description).

Result:
SYNCERR_NONE,
SYNCERR_FILE_ALREADY_EXISTS
SYNCERR_FILE_TOO_MANY_FILES
SYNCERR_REMOTE_BAD_ARG

Description: Creates a new database on the Palm OS device with the name specified in the `CDbCreateDB` structure.

```
class CDbCreateDB
{
public:
  BYTE    m_FileHandle;
```

```
        // Upon return gets filled in by SyncMgr.Dll
    DWORD m_Creator;
        // Supplied by caller, obtained from DbList
    eDbFlags  m_Flags;
        // Supplied by caller, Res/Rec/RAM
    BYTE      m_CardNo;
        // Supplied by caller, target card #
    char      m_Name[DB_NAMELEN];
        // Supplied by caller, target DBase Name
    DWORD     m_Type;
        // for example sysFileTApplication
    WORD      m_Version;
};
```

Upon success, the structure member `m_FileHandle` contains a valid file handle to access the new remote database. When finished using this new handle, the application has to call `SyncCloseDB` to close the handle.

Before calling this function, you have to fill out some of the structure members which influence the newly created database.

- The `m_Flags` member may contain either of the following `eDbFlags` values:
  - `eRecord` indicates a record-oriented database (holding data records)
  - `eResource` indicates a resource-oriented database (usually storing code).
- The structure member m_Type must contain the hexadecimal values for the characters that indicate the type of the database being created, for example:
  - `sysFileTApplication` contains application resources such as executable code.
  - other values are defined by the application and must be mixed case or upper case four-byte values.

SyncDeleteDB

Purpose Delete a database

Prototype
```
long SyncDeleteDB (char* pName, int nCardNum)
```

Parameters

Name	Name of database to remove (must be closed).
CardNum	Number of card where database resides.

Result

SYNCERR_NONE
SYNCERR_FILE_NOT_FOUND
SYNCERR_FILE_OPEN

Description

Instructs the Palm OS device to delete the named database from its storage on the specified card number. The database must be closed (not in use).

SyncOpenDB

Purpose

Open a database on the Palm OS device.

Prototype
```
long SyncOpenDB (  char* pName,
                   int nCardNum,
                   BYTE& rHandle,
                   BYTE openMode)
```

Parameters

pname	Name of remote database to open (contained in `CSyncProperties` structure).
cardNum	Number of memory card on which the `CSyncProperties` structure resides (currently, only 0 is supported).
rHandle	Reference to a BYTE that receives the open file handle.
mode	Bit flag that can be a combination of `eDbWrite`, `eDbRead`, and `eDbExclusive`. In `eDbExclusive` mode, no user can access the file.

(XX doc also has eDbShowSecretXX)

Result

SYNCERR_NONE
SYNCERR_FILE_NOT_FOUND
SYNCERR_FILE_NOT_OPEN
SYNCERR_FILE_OPEN

Description
Opens a database on the Palm OS device for read/write/exclusive access. The name of the database to open is provided to the conduit as part of the `CSyncProperties` structure. Upon successful return, `rHandle` will contain a numeric file handle that should be used in all subsequent file I/O operations.

SyncReadDBAppInfoBlock

Purpose
Locate and retrieve information.

Prototype
```
long SyncReadDBAppInfoBlock (  BYTE fHandle,
                               CDbGenInfo &rDbInfo)
```

Parameters

fHandle	Open valid file handle.
rDbInfo	Reference to a `CDbGenInfoStructure` to receive information.

Result
SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEMES

Description
The AppInfoBlock is a generalized way for a Palm OS device application to store application-specific information in a database. This call instructs the Palm OS device to locate and retrieve the information stores it in the passed `CDbGenInfo` structure.

```
class CDbGenInfocode {
public:
                // Name of remote database file
  char   m_FileName[DB_NAMELEN];
                //Length of m_pBytes buffer
                allocated by the caller.
  WORD   m_TotalBytes;
                // Byte length of 'pBytes'
  WORD   m_BytesRead;
                // Inbound byte count
  BYTE * m_pBytes;
```

```
};
```

The calling client conduit library must allocate enough memory in the general data area to hold the information returned.

If the m_BytesRead value is > m_TotalBytes, then m_pBytes has not been touched. The caller should reallocate m_pBytes to be at least m_BytesRead and make the call again.

If m_BytesRead <= m_TotalBytes then it is the total number of bytes read into m_pBytes.

It is in place to facilitate trading of database-specific information which may assist in the synchronization process. Enough memory (less than 1K) must be preallocated on the incoming pointer by the calling conduit library to hold the response data returned by the devise (and placed in the `m_pBytes` member).

The built-in applications on the device store categories in `AppInfoBlock`. See the *Developing Palm OS Applications* documentation set for more information.

SyncReadDBSortInfoBlock

Purpose Read database information from the device.

Prototype

```
long SyncReadDBSortInfoBlock(  BYTE fHandle,
                               CDbGenInfo &rDbInfo)
```

Parameters

fHandle — Database file handle from an open or create call.

rDbInfo — Reference to a `CDbGenInfoStructure` to receive remote sort information.

Result

SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEME

Description This function lets you read database information from the device, storing it in the `CDbGenInfocode` class.

```
class CDbGenInfocode {
public:
  char   m_FileName[DB_NAMELEN];
                // Name of remote database file
              ??header conflict: NOT USE in doc
  WORD   m_TotalBytes;
                // Byte length of 'pBytes'
  WORD   m_BytesRead;
                // Inbound byte count
  BYTE * m_pBytes;
};
```

The calling client conduit library must preallocate enough memory onto the member `m_pBytes` to hold the incoming reply data. Upon return, the member `m_BytesRead` holds the number of bytes actually transferred to the `m_pBytes` buffer.

This function provides a way to exchange a block of information attached to a database on the device. This function is not required; conduits may or may not use it.

SyncResetSyncFlags

Purpose Reset flags of all open database records that is, clear dirty and archived flags for the whole database.

Prototype `long SyncResetSyncFlags (BYTE fHandle)`

Parameters fHandle Database file handle from an open or create call.

Result SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN.

Description Instructs the Palm OS device to scan all the records of the open database and clears dirty and archived flags. This may or may not be applicable for every conduit.

Applications typically call this function before closing the database.

SyncWriteDBAppInfoBlock

Purpose
Write information to the device.

Prototype
```
long SyncWriteDBAppInfoBlock ( BYTE fHandle,
                               CDbGenInfo rDbInfo)
```

Parameters

fHandle	Database file handle from an open or create call.
rDbInfo	Reference to a `CDbGenInfoStructure` for a remote write.

Result
SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEME.

Description
Instructs the device to write the information in the passed structure to the device's permanent storage associated with the open file handle.

```
class CDbGenInfocode {
public:
  char   m_FileName[DB_NAMELEN];
                // Name of remote database file
              ??header conflict: NOT USE in doc
  WORD   m_TotalBytes;
                // Byte length of 'pBytes'
  WORD   m_BytesRead;
                // Inbound byte count
  BYTE * m_pBytes;
};
```

The structure member `m_TotalBytes` should contain the number of bytes within the `m_pBytes` buffer to actually write to the device.

SyncWriteDBSortInfoBlock

Purpose
Write information to the device.

Prototype
```
long SyncWriteDBSortInfoBlock (BYTE fHandle,
                               CdBGenInfo *pDbInfo)
```

Parameters

fHandle — Database file handle from an open or create call.

rDbInfo — Reference to a `CDbGenInfo` structure containing remote sort information.

Result

SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM

Description

Instruct the device to write the information stored in the passed structure to the device's permanent storage associated with the open file handle. The structure member `m_TotalBytes` should contain the number of bytes within the `m_pBytes` buffer to actually write to the device.

```
class CDbGenInfocode {
public:
  char   m_FileName[DB_NAMELEN];
                 // Name of remote database file
               ??header conflict: NOT USE in doc
  WORD   m_TotalBytes;
                 // Byte length of 'pBytes'
  WORD   m_BytesRead;
                 // Inbound byte count
  BYTE * m_pBytes;
};
```

Record-Oriented Calls

The record-oriented APIs are used to pass the representation of a record (which resides in a database file) between the PC and Pilot. Because one primary purpose of the SyncManager.DlL is to act as a ??shipping channel?? for byte traffic to the device, there is a need for a generic definition of a structure which should handle any record format. This structure then becomes a parameter in these record-oriented APIs.

For reading records, three different APIs are provided, allowing for:

- Sequential location of the next modified record via SyncReadNextModifiedRec
- Exact record lookup via SyncReadRecordById
- Top to bottom iteration via SyncReadRecordByIndex

The same CRawRecordInfo structure is used in all three APIs. However, different structure fields are used by each call. If a field is commented "Filled in by Pilot," the device supplies the data for it.

The record-oriented API provides these calls:

- SyncDeleteAllResourceRec
- SyncDeleteRecord
- SyncDeleteResourceRec
- SyncGetDBRecordCount
- SyncPurgeAllRecs
- SyncReadNextModifiedRec
- SyncReadRecordById
- SyncReadRecordByIndex
- SyncReadResRecordByIndex
- SyncWriteRec
- SyncWriteResourceRec

SyncDeleteAllResourceRec

Purpose Delete all resource records from the currently open database on the device.

Prototype `long SyncDeleteAllResourceRec (BYTE fHandle)`

Parameters fHandle Database file handle from an open or create call.

Result SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN
SYNCERR_ROM_BASED
SYNCERR_READ_ONLY.

Description This routine instructs the device to delete all resource records from the currently open resource database. Use this routine on a remote database con-

sisting of resource type records. These records generally consist of code resources, such as an executable program that runs on the device.

SyncDeleteRecord

Purpose Delete a specified record on the device.

Prototype `long SyncDeleteRecord (CRawRecordINfo &rRec)`

Parameters rRec Reference to incoming `CRawRecordInfo` structure.

Result SYNCERR_NONE, SYNCERR_COM_NOT_INIT, SYNCERR_FILE_NOT_OPEN, SYNCERR_RECORD_BUSY, SYNCERR_FILE_NOT_FOUND, SYNCERR_ROM_BASED, SYNCERR_READ_ONLY.

Description Instructs the device to delete the record specified in the structure member `m_RecId` in the open database.

SyncDeleteResourceRec

Purpose Delete the passed resource on the device.

Prototype `long SyncDeleteResourceRec (CRawRecordINfo rRec)`

Parameters rRec Reference to incoming `cRawRecordInfo` structure.

Result SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN
SYNCERR_FILE_NOT_FOUND
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM

Description This routine instructs the device to delete resource identified by its unique ID (passed in the structure member `m_RecIndex`) from the open database. It is not necessary to allocate memory or fill out any structure members other than the first three.

Use this routine on a remote database consisting of resource type records. These records typically consist of code resources such as executable programs which run on the device.

SyncGetDBRecordCount

Purpose
Obtain total record count from currently open device database.

Prototype
```
long SyncGetDBRecordCount (  BYTE fHandle,
                             Word &rCount)
```

Parameters

fHandle	Database file handle from an open or create call.
rCount	Reference to a variable to receive the record count.

Result
SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN

Description
This routine obtains the total record count for the currently open database on the device.

SyncPurgeAllRecs

Purpose
Delete all records from currently open database on device, regardless of status.

Prototype
```
long SyncPurgeAllRecs (BYTE fHandle)
```

Parameters

fHandle	Database file handle from an open or create call.

Result
SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN
SYNCERR_RECORD_BUSY
SYNCERR_ROM_BASE
SYNCERR_READ_ONLY

Description
This routine instructs the device to delete every record from the currently open database, regardless of the current status flags.

SyncPurgeDeletedRecs

Purpose
Delete all records marked "deleted" from currently open database on the device.

Prototype
```
long SyncPurgeDeletedRecs (BYTE fHandle)
```

Parameters
fHandle Database file handle from an open or create call.

Result
SYNCERR_NONE
SYNCERR_FILE_NOT_OPEN
SYNCERR_ROM_BASED
SYNCERR_READ_ONLY
SYNCERR_REMOTE_RECS_NOT_PURGED

Description
This routine instructs the device to delete all records from the currently open database that have their status flags set to delete. When the user deletes a record on the device, the record is marked for deletion but not actually removed from the data file. This allows the conduit program on the PC to delete matching records from the local data file and purge the record after the PC record has been purged.

SyncReadNextModifiedRec

Purpose
Traverse the currently open database on device and return the next modified record.

Prototype
```
long SyncReadNextModifiedRec (
                                CRawRecordInfo &rRec)
```

Parameters
rRect Reference to incoming `CRawRecordInfo` structure.

Result
SYNCERR_NONE
SYNCERR_COM_NOT_INIT
SYNCERR_RECORD_BUSY
SYNCERR_FILE_NOT_FOUND

Description
Instructs the Palm OS device to traverse its currently open database and return the next record it encounters that has been modified since the last synchronization session.

The caller is expected to have allocated enough memory onto the `m_pBytes` pointer of the `CRawRecordInfo` structure to contain a full record's worth of bytes in the reply from the device. The structure member `m_RecSize` is provided so the remote device can indicate the exact number of bytes returned in the reply data.

The `CRawRecordInfo` structure is defined as follows:

```
class CRawRecordInfo
{
public:
  BYTE m_FileHandle;     // Supplied by caller
  DWORD m_RecId;         // Supplied by caller
                           (when appropriate)
  WORD   m_RecIndex;     // Supplied by caller
                           (when appropriate)
  BYTE m_Attribs;        // Filled in by HH
  short m_CatId;         // Filled in by HH
  int m_ConduitId;       // Ignore
  DWORD m_RecSize;       // Filled in by HH
  WORD m_TotalBytes;     // Supplied by caller
  BYTE * m_pBytes;       // Allocated by caller
};
```

SyncReadRecordById

Purpose Search device database for match on a record.

Prototype `long SyncReadRecordById(CRawRecordInfo &rRec)`

Parameters rRec — Reference to incoming `CRawRecordInfo` structure.

Result
SYNCERR_NONE
SYNCERR_COM_NOT_INIT
SYNCERR_RECORD_BUSY
SYNCERR_FILE_NOT_FOUND

Description — This function can be thought of as a seek and find procedure. The device searches its currently open database and looks for a match on the unique record (supplied in the structure member `m_RecId`). Upon successful execution of the routine, the structure member `m_pBytes` contains the raw record body from the device and the structure member `m_RecSize` is updated with the length of the returned record body.

SyncReadRecordByIndex

Purpose — Traverse Palm OS device database.

Prototype

```
long SyncReadRecordByIndex (CRawRecordInfo &rRec)
```

Parameters — rRec: Reference to incoming `CRawRecordInfo` structure.

Result — SYNCERR_NONE, SYNCERR_COM_NOT_INIT, SYNCERR_FILE_NOT_OPEN, SYNCERR_RECORD_BUSY, SYNCERR_FILE_NOT_FOUND.

Description — By iteratively supplying sequential values to the structure member `m_RecIndex`, starting with zero, a conduit can use this function to traverse a Palm OS device database from top to bottom. The structure member `m_RecIndex` can be though of an array offset, in essence accessing a specific record in an open database by its relative offset from the beginning of the file.

The device typically traverses its currently open database from the top and returns the record body located at the `m_RecIndex` position. Upon successful execution of the routine, the structure member `m_pBytes` will contain the raw record body from the device and the structure member `m_RecSize` is updated with the length of the returned record body.

SyncReadResRecordByIndex

Purpose — Traverse the currently open database on the device.

Prototype

```
long SyncReadResRecordByIndex (
                               CRawRecordInfo &rRec,
                               BOOL bBody
```

Parameters

rRec	Reference to incoming `cRawRecordInfo` structure.
bBody	Indicates whether to retrieve the record (TRUE) or not (FALSE). Default is TRUE.

Result

SYNCERR_NONE
SYNCERR_COM_NOT_INIT
SYNCERR_FILE_NOT_OPEN
SYNCERR_RECORD_BUSY
SYNCERR_FILE_NOT_FOUND
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM

Description

This routine provides a mechanism to traverse the currently open resource database on the device from top to bottom. The structure member `m_RecIndex` can be thought of as an array offset, in essence accessing a specific record in an open database by its relative offset from the beginning of the file. On success, the device returns the record body located at the `m_RecIndex` position.

Upon successful execution of this routine, the structure member `m_pBytes` will contain the raw record body from the device and the structure member `m_RecSize` is updated with the length of the returned record body. Use this routine on a remote database consisting of resource type records. These record types generally consist of code resources, such as an executable program which runs on the device, as well as other types of resources like preferences, images, and so on.

SyncWriteRec

Purpose

Instruct the device to write the passed record into the open database.

Prototype

```
long SyncWriteRec (CRawRecordInfo &rRec)
```

Parameters

rRec	Reference to incoming `cRawRecordInfo` structure.

Result

SYNCERR_NONE
SYNCERR_COM_NOT_INIT
SYNCERR_FILE_NOT_OPEN
SYNCERR_RECORD_BUSY
SYNCERR_FILE_NOT_FOUND

SYNCERR_ROM_BASED
SSYNCERR_READ_ONLY

Description
Instructs the device to write the passed record into the open database. The caller must supply either a valid record ID in the member m_RecId or place zero in this member. This instructs the device to append the record as a new record to the open database. The record body is placed in the memory on the pointer `m_pBytes` and should be formatted to match the record layout in the open database on the device.

SyncWriteResourceRec

Purpose
Write the passed resource into the open database.

Prototype
```
long SyncWriteResourceRec (CRawRecordInfo rRec)
```

Parameters
rRec Reference to incoming `CRawRecordInfo` structure.

Result
SYNCERR_NONE
SYNCERR_COM_NOT_INIT
SYNCERR_FILE_NOT_OPEN
SYNCERR_RECORD_BUSY
SYNCERR_FILE_NOT_FOUND
SYNCERR_ROM_BASED
SYNCERR_READ_ONLY
SYNCERR_REMOTESYS
SYNCERR_REMOTE_MEM

Description
This routine instructs the device to write the resource passed in the structure member `m_RecId` into the open database. The record body contained in the memory on the pointer `m_pBytes` is sent as is and should be formatted to match the resource record layout in the currently open database on the device.

Use this routine on a remote database consisting of resource type records. These records typically consist of code resources, such as an executable program which runs on the device, as well as other types of resource like images or preferences.

Utility Calls

The calls provided by the utility API retrieve information on how the remote device is configured. There is also a function that lets the caller obtain the list of files present on any of the memory cards currently present in the device. The API consists of these calls:

- SyncReadDBList
- SyncReadSingleCardInfo
- SyncReadSystemInfo

SyncReadDBList

Purpose Retrieve information about list of databases on Palm OS device.

Prototype

```
long SyncReadDBList (  BYTE cardNo,
                       WORD startIX,
                       BOOL bRam,
                       CDbList* pList,
                       int& rCnt);
```

Parameters

->cardNo	Number of card to search on Palm OS device.
->startIx	Beginning offset of list to search (0-based)
->bRam	If TRUE, search RAM, otherwise search ROM.
<-pList	Preallocated memory to be filled.
<-rCnt	Number of database entries returned.

Result

SYNCERR_NONE
SYNCERR_FILE_NOT_FOUND
SYNCERR_COMM_NOT_INIT
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM

Description This function allows the caller to discover a list of all the databases (both data and program) that reside on a memory card within the Palm OS device. This is analogous to a directory listing on a PC; the result contains both data files and program files.

The `pList` parameter contains an array of the following structure:

```
class CDbList
{
public:
  int      m_CardNum;
  WORD     m_DbFlags;
             // contains Res/Record/Backup/ReadOnly
  DWORD    m_DbType;
  char     m_Name[DB_NAMELEN];
  DWORD    m_Creator;
  WORD     m_Version;
  DWORD    m_ModNumber;
  WORD     m_Index;
  long     m_CreateDate;
  long     m_ModDate;
  long     m_BackupDate;
  BOOL     m_bReadOnly;
  long     m_RecCount;
  long     m_ModRecCount;
};
```

SyncReadSingleCardInfo

Purpose Retrieve information about the specified memory card.

Prototype `long SyncReadSingleCardInfo (CardInfo &rInfo)`

Parameters rInfo Reference to incoming CardInfo structure.

Result SYNCERR_NONE
SYNCERR_COMM_NOT_INIT
SYNCERR_REMOTE_SYS

Description Retrieves information about a memory card. Memory card numbers on the device start at zero. The caller must fill out the first member of the structure `mCardNo` with the number of the memory card it wants to gather data

about (currently, only 0 is supported). When the call returns, the remaining structure members are filled with data.

```
class CCardInfo
{
public:
  BYTE     m_CardNo;
  WORD     m_CardVersion;
  long     m_CreateDate;
  DWORD    m_RomSize;
  DWORD    m_RamSize;
  DWORD    m_FreeRam;
  BYTE     m_CardNameLen;
  BYTE     m_ManufNameLen;
  char     m_CardName[REMOTE_CARDNAMELEN];
  char     m_ManufName[REMOTE_MANUFNAMELEN];
```

SyncReadSystemInfo

Purpose Retrieve information from the Palm OS device.

Prototype `long SyncReadSystemInfo (CSystemInfo &rInfo)`

Parameters rInfo Reference to an incoming `CSystemInfo` structure.

Result SYNCERR_NONE
SYNCERR_COMM_NOT_INIT
SYNCERR_REMOTE_SYS
SYNCERR_LOCAL_BUFF_TOO_SMALL.

Description Instructs the Palm OS device to populate the passed `CSystemInfo` structure:

```
class CSystemInfo
{
public:
  DWORD   m_RomSoftVersion;
```

```
            // Upon return is filled in
  DWORD   m_LocalId;
          // Upon return is filled in
  BYTE    m_ProdIdLength;
          // Upon return is filled in (actual len)
  BYTE    m_AllocedLen;
          // Supplied by caller
  BYTE*   m_ProductIdText;
          // Allocated by caller
};
```

The information includes the revision level of the ROM software, the ID of the device, a string buffer containing product text information. the caller must preallocate memory on the `m_productIdText` pointer before calling this routine, and initialize the `m_AlloceedLen` member with the size of memory preallocated. If not enough memory (or none at all) is preallocated, the function returns with error SYNCERR_LOCAL_BUFF_TOO_SMALL

7

Error Codes

SyncManager Return Codes

SyncManager return codes begin with the hexadecimal value 0x4000 and are returned as long (four byte) values from each of the public function calls. See also SyncMgr.h.
SyncManager return codes are as follows:

Return Code	Meaning
SYNCERR_NONE 0x000	Function completed successfully.
SYNCERR_FILE_NOT_FOUND 0x4003	Database filename could not be found on the Palm OS device.
SYNCERR_FILE_NOT_OPEN 0x4004	Database on the Palm OS device is not currently open, or the handle value is invalid.
SYNCERR_FILE_OPEN 0x4004	Database already open. Cannot be reopened in the current mode.
SYNCERR_RECORD_BUSY 0x4006	Write or delete operation could not be performed on the specified record ID because Palm OS device is already using the record. File is busy or another process is accessing it.
SYNCERR_RECORD_DELETED 0x4007	Could not read or update the record because it not longer exists.
SYNCERR_ROM_BASED 0x4008	Writing a ROM-based database is not allowed.
SYNCERR_READ_ONLY 0x4009	Writing to a read-only database is not allowed.

Return Code	Meaning
SYNCERR_COM_NOT_INIT 0x0A	Failed to create a valid internal communications object.
SYNCERR_FILE_ALREADY_EXISTS	Cannot create the specified database. The file already exists on the Palm OS device.
SYNCERR_FILE_ALREADY_OPEN	Cannot create the specified database. The database is currently open.
SYNCERR_NO_FILES_OPEN	Protocol error.
SYNCERR_BAD_OPERATION	Protocol error.
SYNCERR_REMOTE_BAD_ARG	An invalid structure member was supplied.
SYNCERR_BAD_ARG_WRAPPER	Protocol error.
SYNCERR_ARG_MISSING	Protocol error.
SYNCERR_LOCAL_BUFFER_TOO_SMALL	Insufficient memory was allocated for the incoming record.
SYNCERR_REMOTE_MEM	Memory allocation failed on the Palm OS device. This is a nonfatal memory condition. Either the communications layer or the device application could not perform the operation. This does not necessarily indicate that there is no more memory on the device. Other operations could potentially be performed.
SYNCERR_REMOTE_NO_SPACE	There is no space on the Palm OS device to add records to the database.

SyncManager Fatal Return Codes

For fatal return codes, the high bit of the long value is set, which indicates that the synchronization session has already been halted or is in such a misaligned state that no fur-

ther calls should be made into the SyncManager library. See also syncmgr.h.
The following fatal codes are currently defined:

Return Code	Meaning
SYNCERR_REMOTE_SYS	System failure on the Palm OS device.
SYNCERR_TOO_MANY_FILES	Cannot create; too many files already exist.
SYNCERR_REMOTE_CANCEL_SYNC	User cancelled synch session from device.

SyncManager Base Class Return Codes

The base class returns a base class error if the conduit uses the built-in synchronization logic. Base class returns codes for the Conduit DLLs range from 0x5000 through 0x5FF. See also basemon.h
The following base class errors are currently defined:

Return Code	Meaning
CONDERR_NONE	Function completed successfully.
CONDERR_NO_REMOTE_CATEGORIES CONDERR_FIRST+1	Not currently used.
CONDERR_NO_LOCAL_CATEGORIES CONDERR_FIRST+2	Not currently used.
CONDERR_SAVE_REMOTE_CATEGORIES CONDERR_FIRST+3	Problems opening or creating the remote database.
CONDERR_BAD_REMOTE_TABLES CONDERR_FIRST+4	Not currently used.
CONDERR_BAD_LOCAL_TABLES CONDERR_FIRST+5	Problems appending duplicate record.
CONDERR_BAD_LOCAL_BACKUP CONDERR_FIRST+6	Not currently used.
CONDERR_ADD_LOCAL_RECORD CONDERR_FIRST+7	Problems appending duplicate record.

Return Code	Meaning
CONDERR_ADD_REMOTE_RECORD CONDERR_FIRST+8	Problems adding the remote record.
CONDERR_CHANGE_REMOTE_RECORD CONDERR_FIRST+9	Could not allocate memory for buffer.
CONDERR_RAW_RECORD_ALLOCATE CONDERR_FIRST+0x0A	Memory allocation for a buffer used to hold an incoming raw record from the Palm OS device failed.
CONDERR_REMOTE_CHANGES_NOT_SENT CONDERR_FIRST+0x0B	Failed to send all changes to Palm OS device.
CONCERR_LOCAL_MEMORY_ALLOC_FAILED CONDERR_FIRST+0x0C	Attempt to allocate memory for an AppInfoBlock to be read from device.
CONDERR_CONVERT_TO_REMOTE_CATS CONDERR_FIRST+0x0D	Failure to convert.
CONDERR_CONVERT_TO_LOCAL_CATS CONDERR_FIRST+0x0E	??Failure to convert.
CONDERR_CONVERT_TO_REMOTE_REC CONDERR_FIRST+0x0F	Could not convert to raw record layout.
CONDERR_CONVERT_FROM_REMOTE_REC CONDERR_FIRST+0x010	Could not convert remote record.
CONDERR_REMOTE_RECS_NOT_PURGED CONDERR_FIRST+0x011	Issued during either a `CopyToPC()` or `CopyToHH()` call. After either call, an attempt is made to purge the remote records marked for deletion on the device. If a derived base conduit monitor calls `SyncPurgeAllRecs()` and that call fails, this code is returned.

Return Code	Meaning
CONDERR_BAD_SYNC_TYPE CONDERR_FIRST+0x012	The conduit was initiated to being its operations, but was passed a synchronization action it did not understand. The following valid actions are defined: eFast, eSlow, eHHtoPC, ePCtoHH, eInstall, eBackup
CONDERR_DATE_MOVED CONDERR_FIRST+0x050	Issued by the baseDTLinkConverter in its routine to convert a data field from the device format to the PC. The device allows users to enter dates that precede 1970. On the PC, dates before 1970 are invalid, so the converter moves any dates before 1970 up to 1970. This code is warning, not an error; information is saved in the log when a date conversion takes place.