palmsource™

C/C++ Sync Suite Reference for Macintosh

Palm OS® Conduit Development Kit for Macintosh, Version 4.03

CONTRIBUTORS

Updated by Eric Shepherd and Brent Gossett
Engineering contributions by Cole Goeppinger and Chris Tate

C/C++ Sync Suite Reference for Macintosh
Document Number 3001-003-HW
June 19, 2002
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

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

Table of Contents

About This Document

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

The *C/C++ Sync Suite Reference for Macintosh* provides you with complete reference information for all of the constants, data structures, classes, functions, and methods that you can use to develop conduits.

The sections in this introduction are:

- Conduit Development Kit Documentation
- What this Document Contains
- HotSync Manager Application
- CDK Version Compatibility
- CDK/Handheld Compatibility
- Conventions Used in this Document
- Additional Resources

Conduit Development Kit Documentation

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

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

The following two documents are part of the C/C++ Sync Suite:

Document	Description
C/C++ Sync Suite Companion for Macintosh	An introduction to conduit development that provides an overview of how conduits are used and how to implement them.
C/C++ Sync Suite Reference for Macintosh	This API reference document that contains descriptions of all conduit function calls and important data structures.

For more information about programming for the Palm OS platform, see the following Palm OS documents, which are part of the Palm OS Software Development Kit:

Document	Description
Palm OS Programmer's API Reference	An API reference document that contains descriptions of all Palm OS function calls and important data structures.
Palm OS Programmer's Companion	A guide to application programming for the Palm OS. This volume contains conceptual and "how-to" information that compliments the Reference.
Palm OS Programming Development Tools Guide	A guide to writing and debugging Palm OS applications with the various tools available.

What this Document Contains

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

- Chapter 1, "Conduit API." Describes the conduit API functions, which are callback functions that your conduit must implement to communicate with the HotSync® Manager application.

- Chapter 2, "Sync Manager API." Describes the Sync Manager functions, constants, classes, and data structures, which you can use to exchange data between a desktop computer and a handheld.
- Chapter 3, "HotSync Log API." Describes the HotSync log functions, which you can use to access the HotSync log.
- Chapter 4, "Expansion Manager API." Describes the Expansion Manager functions, constants, and data structures, which you can use to get information about expansion cards in the handheld.
- Chapter 5, "Virtual File System Manager API." Describes the VFS Manager functions, constants, and data structures, which you can use to access file systems on expansion cards in the handheld.
- Chapter 6, "User Manager API." Describes the User Manager functions, constants, and data struc tures, which let you find information about users and queue files to be installed onto their handhelds.
- Appendix A, "Private Functions."Summarizes the private system functions that you may see named in source code but cannot use in your conduits.

HotSync Manager Application

On Macintosh desktop computers, the HotSync Manager Application is implemented in three components:

- The HotSync Manager is the executable that provides configuration and logging facilities for synchronization operations.
- The Serial Monitor is a background application that watches the serial port for the wakeup packet from the handheld and then passes control to the Conduit Manager.
- The Conduit Manager manages the synchronization process and the execution of the individual conduits.

This book uses the term "the HotSync Manager application" to refer to these three components as a single entity.

CDK Version Compatibility

Each Palm Powered handheld ships with a specific version of the HotSync Manager, which is the application that runs on the Macintosh desktop computer and controls synchronization operations.

Each version of the HotSync Manager includes a version of the interface to an important data access library used by conduits, which is called the Sync Manager API.

You can use this conduit development kit to develop conduits for the Macintosh HotSync Manager version 2.0, and for Sync Manager API versions through version 2.1.

The table below shows the mapping from HotSync Manager versions to Sync Manager API versions.

Palm Desktop Version	HotSync Manager Version	Sync Manager API Version
2.0 ?	2.0	2.1
2.6.3	2.0	2.2
4.0	3.0	2.3

CDK/Handheld Compatibility

You can use this CDK to develop conduits for all Palm Powered handhelds, including those developed by Palm OS licensees and OEM partners.

Conventions Used in this Document

This guide uses the following typographical conventions:

This style...	**Is used for...**
`fixed width font`	Code elements such as function, structure, field, bitfield.
bold	Emphasis.
blue and underlined	Hot links.
▼	New functions added since the last release of the CDK.
-->	Parameter is passed in to a function.
<--	Parameter is passed out of a function.
<->	Parameter passed in and out of a function.

Additional Resources

- Documentation

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

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

- Training

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

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

- Knowledge Base

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

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

1

Conduit API

This chapter describes the conduit API functions, which are callback functions that your conduit must implement to communicate with the HotSync® Manager application.

Conduit API Constants

This section describes the constants that you can use with the conduit API functions.

Synchronization Property Constants

You use the synchronization property constants to specify the synchronization properties for your conduit.

Synchronization Type (eSyncTypes) Constants

The synchronization type constants are used to tell your conduit which type of synchronization operation to perform. You also use one of these values in an object of The CSyncPreference Class to inform the HotSync Manager application of your conduit's mode.

```
enum eSyncTypes{eFast,
                eSlow,
                eHHtoPC,
                ePCtoHH,
                eInstall,
                eBackup,
                eDoNothing,
                eProfileInstall } ;
```

`eFast`	Perform a fast synchronization: only records that have been added, archived, deleted, or modified are exchanged between the handheld and the desktop computer.

`eSlow`	Perform a slow synchronization: every record is read from the handheld and compared to the corresponding record on the desktop computer. Used when the handheld has been synchronized with multiple computers.
`eHHtoPC`	Perform a restore from the handheld: overwrite the desktop database with the database from the handheld.
`ePCtoHH`	Perform a restore from the desktop computer: overwrite the database on the handheld with the database on the desktop computer.
`eInstall`	Install new applications from the desktop computer to the handheld.
`eBackup`	Perform a backup of the databases on the handheld to the desktop computer.
`eDoNothing`	The conduit does not exchange data between the handheld and the desktop computer; however, the conduit is loaded and can set flags or log messages.
`eProfileInstall`	Perform a profile download. A profile is a special user account that you can set up on the desktop computer that downloads data to a handheld, erasing all information on the handheld and leaving it without a user ID.

NOTE: If the synchronization type is `eProfileInstall`, conduits are supposed to perform a synchronization in which the desktop overrides the handheld (`ePCtoHH`). The Windows version of the HotSync Manager application substitutes the properties, but the Macintosh version does not.

Conduits running on the Macintosh with synchronization type `eProfileInstall` must include the `ePCtoHH` flag in the `m_SyncType` field, and must substitute before processing the data.

First Synchronization (eFirstSync) Constants

You use the first synchronization constants to determine whether the handheld has previously been synchronized with the desktop computer.

```
enum eFirstSync{
                eNeither,
                ePC,
                eHH } ;
```

`eNeither`	The handheld has been synchronized before with this desktop computer: the handheld user ID matches a user ID on the desktop computer.
`ePC`	The handheld has been synchronized before, but has never been synchronized on the desktop computer: the handheld has a user ID that does not match a user ID on the desktop computer.
`eHH`	The handheld has not been synchronized before and thus does not have a user ID. This might indicate a handheld that was recently reset or installed with a profile.

Synchronization Connection Type (eConnType) Constants

You use the synchronization connection type constants to determine how the handheld is connected to the desktop computer.

```
enum eConnType  { eCable, eModemConnType };
```

`eCable`	The handheld is connected with a fast connection, such as a direct cable.
`eModemConnType`	The handheld is connected with a slow connection via a modem.

Synchronization Preferences (eSyncPref) Constants

You use the synchronization preferences constants to specify whether the user preferences apply temporarily or permanently.

NOTE: The synchronization preference values are not currently used in the Macintosh CDK.

```
enum eSyncPref{
               eNoPreference,
               ePermanentPreference,
               eTemporaryPreference };
```

`eNoPreference`	Not specified.
`ePermanentPreference`	The preferences are permanent.
`eTemporaryPreference`	The preferences are temporary and only apply to the next synchronization operation.

Conduit Configuration Classes

This section describes the classes that you use with the conduit API functions.

The CSyncPreference Class

```
class CSyncPreference
{
public:
  FSSpec          m_UserDirFSSpec;
  char            m_Registry[BIG_PATH];
  HKEY            m_hKey;
  eSyncPref       m_SyncPref;
  eSyncTypes      m_SyncType;
  DWORD           m_dwReserved;
};
```

CSyncPreference Data Members

--> `m_userDirFSSpec`	The directory in which data files are to be stored.
--> `m_Registry`	Not used for Macintosh conduits.
--> `m_hKey`	Not used for Macintosh conduits.
--> `m_SyncPref`	The synchronization preference value. Use one of the values described in Synchronization Preferences (eSyncPref) Constants.
--> `m_SyncType`	The current synchronization type. Use one of the values described in Synchronization Type (eSyncTypes) Constants.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before using this object.

The CSyncProperties Class

You can use objects of the `CSyncProperties` class to store information about the properties of the current conduit's synchronization operations. This class is used with several methods and functions that are described in other chapters in this book.

```
class CSyncProperties
{
public:
  eSyncTypes    m_SyncType;
  FSSpec        m_UserDirFSSpec;
  char          m_LocalName[BIG_PATH];
  char          m_UserName[BIG_PATH];
  char**        m_RemoteName[SYNC_DB_NAMELEN];
  CDbListPtr*   m_RemoteDbList;
  int           m_nRemoteCount;
  DWORD         m_Creator;
  WORD          m_CardNo;
  DWORD         m_DbType;
  DWORD         m_AppInfoSize;
  DWORD         m_SortInfoSize;
  eFirstSync    m_FirstDevice;
  eConnType     m_Connection;
  char          m_Registry[BIG_PATH];
  HKEY          m_hKey;
  DWORD         m_dwReserved;
};
```

CSyncProperties Class Members

--> `m_SyncType`	The current synchronization type. Use one of the values described in Synchronization Type (eSyncTypes) Constants.
--> `m_UserDirFSSpec`	The directory in which data files are to be stored.
--> `m_LocalName`	The file names on the desktop computer.
--> `m_UserName`	The name of the user.
--> `m_RemoteName`	An array of the name of the databases on the handheld.

--> `m_RemoteDbList`	A list of the databases on the handheld.
--> `m_nRemoteCount`	The number of database files on the handheld.
--> `m_Creator`	The creator ID of the database on the handheld.
--> `m_CardNo`	The number of the memory card on the handheld on which databases are stored. This is used to create databases on the handheld.
--> `m_DbType`	The database type, which is used to create databases on the handheld.
--> `m_AppInfoSize`	The size of the application information block, which is stored in this object for convenience.
--> `m_SortInfoSize`	The size of the application sort information block, which is stored in this object for convenience.
--> `m_FirstDevice`	Specifies whether the handheld has previously been synchronized with the desktop computer. Use one of the values described in First Synchronization (eFirstSync) Constants
--> `m_Connection`	The connection type for this synchronization. Use one of the values described in Synchronization Connection Type (eConnType) Constants.
--> `m_Registry`	Not used for Macintosh conduits.
--> `m_hKey`	Not used for Macintosh conduits.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before using this object.

Conduit API Function Summary

This section describes the conduit API functions, each of which is described in the remainder of this chapter. These are callback functions that HotSync Manager calls in your conduit. Which callbacks you implement depends on whether your conduit is bundled or not.

You must provide an implementation of each of these functions:

- ConfigureConduit
- GetConduitVersion

If your conduit isn't bundled, you must provide:

- GetActionString
- GetConduitName
- OpenConduit

If your conduit is bundled, you must provide:

- CopyActionStringAsCFStringRef
- CopyConduitNameAsCFStringRef
- OpenConduitCarbon

ConfigureConduit

Purpose Displays and handles a modal dialog box for configuring the conduit.

Prototype `long ConfigureConduit(CSyncPreference& syncPrefs);`

Parameters

--> `syncPrefs`	An object of The CSyncPreference Class, which contains information for your conduit.

Result If successful, returns `0`.

If unsuccessful, returns a non-zero error code value. Use a standard Macintosh toolbox error code or one of the Sync Manager error codes, which are shown in Table 2.1.

Comments The HotSync Manager application calls the `ConfigureConduit` function when a user decides to configure your conduit.

If your conduit does not use a configuration dialog box, you must set a flag in the `'Cinf'` resource for your conduit, which indicates that your conduit has no user interface. Your implementation of this function then does nothing except return a value of `0`.

Your implementation of the `ConfigureConduit` function displays a modal dialog box that allows the user to configure your conduit. The resources for the dialog box are stored in the resource fork of your conduit. When the user dismisses the dialog box, you need to save the settings in your conduit's settings file.

The template conduit that is shipped with the CDK provides an example of the appropriate look and feel for a conduit configuration dialog box, and shows how to determine the location of your conduit's settings file.

NOTE: PalmSource recommends that you use the method shown in the template conduit for storing your settings; this standardizes how conduits work on the Macintosh. PalmSource also recommends that your dialog box use the look and feel of the template conduit's configuration dialog box.

GetActionString

Purpose Called to retrieve a text string that can be displayed to show the current settings or state of your conduit.

Prototype

```
long GetActionString(CSyncPreference& syncPrefs,
char* ioActionStr, WORD inStrLen);
```

Parameters

--> `syncPrefs`	An object of The CSyncPreference Class, which contains information for your conduit.
--> `ioActionStr`	A character buffer. Upon return, contains the action string, stored as a C string.
--> `inStrLen`	The maximum number of bytes that can be stored in the string buffer.

Result If successful, returns `0`.

If unsuccessful, returns a non-zero error code value. Use a standard Macintosh toolbox error code or one of the Sync Manager error codes, which are shown in Table 2.1.

Comments The HotSync Manager application calls the `GetActionString` function to retrieve a current action information display string for your conduit. Your implementation must fill in the buffer with a C string that describes the current state or settings.

The returned string is displayed in the HotSync Manager application's Actions window. For proper alignment in the window, the string should be a maximum of 45 characters long.

If your conduit does not have any settings, implement this function to return a standard string, such as "Synchronize" or "Normal."

NOTE: This function should only be implemented if your conduit isn't bundled.

GetConduitName

Purpose Called to retrieve the name to use for your conduit when displaying messages to the user.

Prototype `long GetConduitName(char* name, WORD nLen);`

Parameters

<-- `name`	A character buffer. Upon return, contains the name of your conduit.
--> `nLen`	The maximum number of bytes that can be stored in the name character buffer.

Result If successful, returns `0`.

If unsuccessful, returns a non-zero error code value. Use a standard Macintosh toolbox error code or one of the Sync Manager error codes, which are shown in Table 2.1.

Comments HotSync Manager calls the `GetConduitName` function to retrieve the name of your conduit. Your implementation must fill in the name buffer with your conduit's name.

NOTE: This function should only be implemented if your conduit isn't bundled.

GetConduitVersion

Purpose Called to retrieve the version number of your conduit.

Prototype `DWORD GetConduitVersion();`

Parameters None.

Result The version number of your conduit, packed into the DWORD result as follows:

The major version number is HIBYTE(LOWORD)

The minor version number is LOBYTE(LOWORD)

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

OpenConduit

Purpose Starts the synchronization process for the conduit by allocating and engaging the synchronizer object.

Prototype
```
long OpenConduit(PROGRESSFN pFn, CSyncProperties&
syncPrefs);
```

Parameters

--> `pFn`	A pointer to a function that the HotSync Manager application can call to report the progress.
--> `syncPrefs`	An object of The CSyncProperties Class, which specifies information about the properties of your conduit's current synchronization operations

Result If successful, returns `0`.

If unsuccessful, returns a non-zero error code value. Use a standard Macintosh toolbox error code or one of the Sync Manager error codes, which are shown in Table 2.1.

Comments HotSync Manager calls the `OpenConduit` function to begin the process of synchronizing data between the desktop computer and the handheld. Your implementation of this function needs to perform operations such as the following:

- create a new instance of your synchronizer class
- engage your monitor by calling its `Synchronize` method
- delete the synchronizer object

NOTE: This function should only be implemented if your conduit isn't bundled.

CopyActionStringAsCFStringRef

Purpose Called to retrieve a text string that can be displayed to show the current settings or state of your conduit. Required only if your conduit is bundled.

Prototype

```
long
CopyActionStringAsCFStringRef(CSyncPreference&
syncPrefs, CFStringRef *ioActionStr);
```

Parameters

--> `syncPrefs`	An object of The CSyncPreference Class, which contains information for your conduit.

--> `ioActionStr`	A `CFStringRef` buffer pointer. Upon return, contains the a reference to the action string.

If successful, returns `0`.

If unsuccessful, returns a non-zero error code value. Use a standard Macintosh toolbox error code or one of the Sync Manager error codes, which are shown in Table 2.1.

Comments HotSync Manager calls the `CopyActionStringAsCFStringRef` function in bundled conduits to retrieve a current action information display string for your conduit. Your implementation must fill in the buffer with a reference to a `CFString` that describes the current state or settings.

The returned string is displayed in the HotSync Manager application's Actions window. For proper alignment in the window, the string should be a maximum of 45 characters long.

If your conduit does not have any settings, implement this function to return a standard string, such as "Synchronize" or "Normal."

NOTE: This function should only be implemented if your conduit is bundled.

CopyConduitNameAsCFStringRef

Purpose Returns a copy of your conduit's name as a `CFStringRef`. Required only if your conduit is bundled.

Prototype

```
long CopyConduitNameAsCFStringRef(CFStringRef
*name);
```

Parameters

--> `name`	A pointer to a `CFStringRef` into which a reference to the conduit's name will be placed.

Result If successful, returns `0`.

If unsuccessful, returns a non-zero error code value. Use a standard Macintosh toolbox error code or one of the Sync Manager error codes, which are shown in Table 2.1.

Comments HotSync Manager calls the `CopyConduitNameAsCFStringRef` function to retrieve the name of your conduit. Your implementation must fill in the `name` buffer with a reference to your conduit's name.

NOTE: This function should only be implemented if your conduit is bundled.

OpenConduitCarbon

Purpose Starts the synchronization process for the conduit by allocating and engaging the synchronizer object. Required only if your conduit is bundled.

Prototype
```
long OpenConduitCarbon(PROGRESSFNCARBON pFn,
CSyncProperties& syncPrefs);
```

Parameters

--> `pFn`	A pointer to a function that the HotSync Manager application can call to report the progress.
--> `syncPrefs`	An object of The CSyncProperties Class, which specifies information about the properties of your conduit's current synchronization operations

Result If successful, returns `0`.

If unsuccessful, returns a non-zero error code value. Use a standard Macintosh toolbox error code or one of the Sync Manager error codes, which are shown in Table 2.1.

Comments This function is optional; you only need to provide it if your Carbonized conduit is bundled. If this function doesn't exist, HotSync Manager will call `OpenConduit` instead.

HotSync Manager calls the `OpenConduitCarbon` function to begin the process of synchronizing data between the desktop computer and the handheld. Your implementation of this function needs to perform operations such as the following:

- create a new instance of your synchronizer class
- engage your monitor by calling its `Synchronize` method
- delete the synchronizer object

NOTE: This function should only be implemented if your conduit is bundled.

2

Sync Manager API

Use the Sync Manager with your conduit to exchange data with Palm Powered™ handhelds that are connected to the desktop computer. Handhelds can be connected to the desktop computer with a cradle and a serial cable, or modem connection. The Sync Manager handles the actual handheld communications and provides functions to receive data from and send data to databases on the handheld.

Sync Manager Versions

The Sync Manager continues to evolve with new functions and new versions of older functions. Each version of the Sync Manager API has a major version number and a minor version number. You can determine the version of the Sync Manager API that you are using by calling the `SyncGetAPIVersion` function.

The Sync Manager maintains backward compatibility within a major version. The Sync Manager minor version number changes when new functions are added or bug repairs are completed. This document includes version information for each function.

NOTE: Conduits developed with the Conduit Development Kit can be used on desktop computers that are running version 2.0 or later of the Sync Manager API. Your conduits will not operate on desktop computers that have earlier versions of the Sync Manager API.

If your conduit depends on functions that are available only in certain versions of the Sync Manager API, you need to determine the version of the Sync Manager API with which you are dealing on a specific installation. To do so, call the `SyncGetAPIVersion` function, which returns both the major version number and minor version number of the Sync Manager API on the desktop computer.

For example, if your conduit depends on a function that was added in version 2.1 of the Sync Manager API, you need to call the `SyncGetAPIVersion` function and then verify that the major number is `2` or greater and that the minor version number is `1` or greater.

HotSync Manager and Sync Manager API Versions

You need to know that the version of the HotSync® Manager application running on a specific desktop computer is not necessarily the same as the version of the Sync Manager API installed on that system. The calls that your conduit can make are dependent on the Sync Manager API version installed on the desktop computer.

The table below shows the mapping from HotSync Manager versions to Sync Manager API versions.

Palm Desktop Version	HotSync Manager Version	Sync Manager API Version
2.0.X	2.0	2.1
2.6.3	2.0	2.2
4.0	3.0	2.3

Using the Sync Manager

The HotSync Manager application calls each conduit that is installed on the user's desktop computer in sequence. The HotSync Manager application calls an entry point, such as the OpenConduit function, in your conduit, which allows your conduit to begin its operations. You then call Sync Manager functions to exchange data with handheld.

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

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

Sync Manager Error Checking

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

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

If you fail to check the error code and then use data values returned by a function that returned an error, you can easily corrupt your database.

Classes and Structures in the Sync Manager

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

The Sync Manager and Performance

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

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

- Avoid reading a record or information block more than once from the handheld database. You can cache information on the desktop computer to preclude the need to access the same information more than once.

- When you are reading information from a database, allocate a buffer that is large enough to store the largest possible record size in that database. This is significantly faster than making one call to determine the data size and another call to actually read the data.

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

- SyncGetAPIVersion
- SyncHHToHostWord
- SyncHostToHHDWord
- SyncHostToHHWord
- SyncYieldCycles

Handheld Database Types

You can create or use two different kinds of databases on handhelds:

- record databases store application data in records
- resource databases store resources; for example, an application that runs on the handheld is a resource database consisting of code, user interface, and other resources.

Both database types use a similar layout format, and both provide interface functions for accessing and modifying database entries.

Handheld Record Database Layout

Each handheld database is stored in memory as a file header followed by a sequence of records or resources.

Record and resource databases are used differently and have different attributes. However, they do have similar structures, in that each manages multiple blocks of arbitrary data, and each has a header block.

The file header section of each database contains the following information about the database:

- its name
- its creator
- its type
- the number of records
- the last synchronization date
- an optional, variable-length application information block
- an optional, variable-length sorting information block

Each record in a record database consists of the following information:

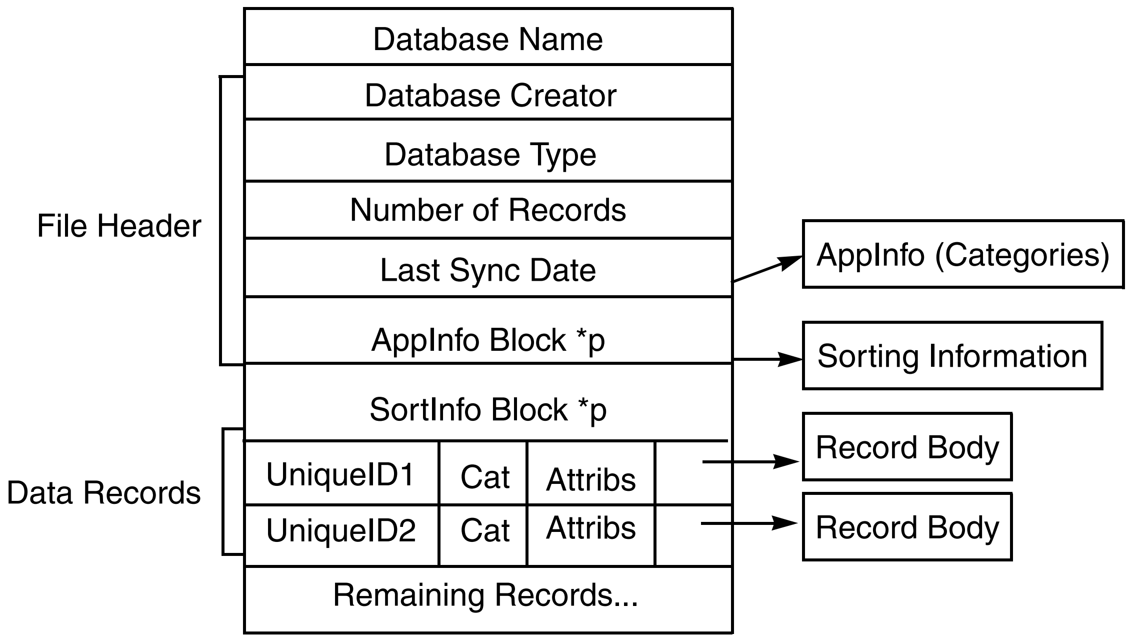
- the record ID
- the record category index
- record attributes
- record data

Each resource in a resource database consists of the following information:

- the resource type
- the resource ID
- record data

Figure 2.1 shows the layout of a handheld record database. For more detailed information about database layout and the data structures used by the built-in applications, see the Palm OS SDK documentation.

Figure 2.1 Handheld record database layout



The File Header Information Blocks

The database file header contains two variable length blocks that you can use to exchange information with the database:

- the application information block, referred to as `AppInfo` in Figure 2.1, is a block of arbitrary information that an application can store in its database. Several of the built-in applications use this block to store the category strings associated with the database. You can use the SyncReadDBAppInfoBlock and SyncWriteDBAppInfoBlock functions to access this block in a database on the handheld.
- the sorting information block, referred to as `SortInfo` in Figure 2.1, is used by applications to store information about how records in the database are sorted. The built-in applications do not currently store any information in this block. You can use the SyncReadDBSortInfoBlock and SyncWriteDBSortInfoBlock functions to access this block in a database on the handheld.

Database Record Attributes

Each database record on the handheld contains a byte of attribute flag bits, as described in the Record Attributes (eSyncRecAttrs) Constants section. You can use these bits to determine record conditions including whether the record has been marked as modified, marked for archiving, marked for deletion, or marked as private.

Reading Handheld Database Records and Resources

The Sync Manager provides a number of functions for retrieving records and resources from a database on the handheld. Each of these functions reads the record or resource into an object of The CRawRecordInfo Class.

You must fill in certain raw record object fields before calling each function. The function retrieves the record from the handheld and fills in other fields with the record information. The fields required for each function are described in the documentation for each function.

You can use the following Sync Manager functions to read records or resources from a database on the handheld:

- the SyncReadRecordById retrieves a record that you specify by record ID.
- the SyncReadRecordByIndex retrieves a record that you specify by record index.
- the SyncReadResRecordByIndex retrieves a resource that you specify by index from a resource database.

Iterating Through a Database

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

- the SyncReadNextRecInCategory retrieves the next record in a certain record category.

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

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

Modifying a Database While Iterating

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

Writing Handheld Database Records and Resources

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

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

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

Deleting Records in Handheld Databases

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

Developers have been confused in the past because some of the deletion function names begin with the `Delete` prefix and others with the `Purge` prefix. All of these functions delete objects.

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

- the SyncDeleteRec function deletes a record from a record database.
- the SyncDeleteResourceRec function deletes a specific resource from a resource database.

Deleting Multiple Records From a Handheld Database

You can delete multiple records by calling one of the following Sync Manager functions:

- the SyncPurgeAllRecs function deletes and removes all records in a record database on the handheld.
- the SyncPurgeAllRecsInCategory function deletes and removes all of the records in a specific category from a record database on the handheld.
- the SyncPurgeDeletedRecs function deletes and removes all of the records in a record database that have previously been marked as deleted or archived.
- the SyncDeleteAllResourceRec function deletes and removes all resources from a resource database.

Maintaining a Connection With the Handheld

If your conduit is performing any time-consuming operations between calls that exchange data with the handheld, you need to ping the handheld to keep the connection alive. For example, you might be retrieving data from a network database that takes considerable time to access. To ping the handheld, you call the SyncYieldCycles function:

```
SyncYieldCycles(WORD wMaxMilliSecs)
```

The `wMaxMilliSecs` parameter specifies the maximum number of milliseconds to spend servicing events. You should set this value to 1 millisecond, as any larger value is most likely a waste of time.

You need to call the SyncYieldCycles function as often as possible. Calls to this function do not appreciably affect the performance of your conduit.

There are several reasons why your conduit needs to call SyncYieldCycles frequently:

- to ensure that the HotSync Manager application progress dialog box remains active; otherwise, the user could press the Cancel button without effect.
- to maintain the HotSync Manager application progress dialog arrow animation, which provides visual feedback to the user that the system is active.
- to keep the connection with the handheld alive when your conduit is performing time-consuming activities; for example, if your conduit is accessing data from a server or retrieving data from a very large database. In these cases, you might not communicate with the handheld for a long enough period of time and cause it to terminate the connection.

Versions of the HotSync Manager application earlier than version 2.1 are single-threaded, which makes calling this function vital to maintain the connection. Version 2.1 of the HotSync application added a second thread that maintains the connection during synchronization operations; however, you must still call the `SyncYieldCycles` function to process user-interface events and update the HotSync display.

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

Sync Manager Constants

This section describes the constants that you use with the Sync Manager functions.

General Constants

These are the general-purpose constants for use with the Sync Manager API.

`BIG_PATH`	The maximum size of a file path specification.
`SYNC_DB_NAMELEN`	The maximum size of a handheld database name, including the null terminator character. This constant replaces the older constant `DB_NAMELEN`.
`SYNC_MAX_HH_LOG_SIZE`	The maximum size of the HotSync log on the handheld.
`SYNC_MAX_PROD_ID_SIZE`	The number of bytes in the product ID buffer.
`SYNC_MAX_USERNAME_LENGTH`	The maximum length of a user name (not including the null terminator character) on the handheld.
`SYNC_REMOTE_CARDNAME_BUF_SIZE`	The buffer size for the name of the memory card on the handheld. This constant replaces the older constant `REMOTE_CARDNAMELEN`.
`SYNC_REMOTE_MANUFNAME_BUF_SIZE`	The buffer size for the manufacturer name on the handheld. This constant replaces the older constant `REMOTE_MANUFNAMELEN`.
`SYNC_REMOTE_PASSWORD_BUF_SIZE`	The buffer size for the password on the handheld. This constant replaces the older constant `PASSWORD_LENGTH`.

`SYNC_REMOTE_USERNAME_BUF_SIZE`	The buffer size for the user name on the handheld. This constant replaces the older constant `REMOTE_USERNAME`.

Database Flag (eDbFlags) Constants

You can combine the database flag constants together to specify information about a database. Each flag indicates a property or condition of the database.

```
enum eDbFlags {
  eRecord             = 0x0000,
  eResource           = 0x0001,
  eReadOnly           = 0x0002,
  eAppInfoDirty       = 0x0004,
  eBackupDB           = 0x0008,
  eOkToInstallNewer   = 0x0010,
  eResetAfterInstall  = 0x0020,
  eCopyPrevention     = 0x0040,
  eStream             = 0x0080,
  eHidden             = 0x0100,
  eLaunchableData     = 0x0200,
  eRecyclable         = 0x0400,
  eBundle             = 0x0800,
  eOpenDB             = 0x8000
};
```

Note that the `eRecord` and `eResource` flags are mutually exclusive and that you must specify exactly one of them when creating a database.

`eRecord`	When this flag is set, indicates that the database is a record database. This is the default value.
`eAppInfoDirty`	When this flag is set, indicates that the application information block has been modified.
`eBackupDB`	When this flag is set, indicates that the database is to be backed up to the desktop if no application-specific conduit has been supplied.

`eOkToInstallNewer`	When this flag is set, indicates that the backup/restore conduit can install a newer version of the database with a different name if the current database is currently opened. For example, the Graffiti® 2 shortcut database is updated in this manner.
`eCopyPrevention`	When this flag is set, indicates that the database is not to be copied or beamed to other handhelds.
`eStream`	When this flag is set, indicates that the database is used for file stream implementation.
`eHidden`	When this flag is set, indicates that this database should generally be hidden from view. It is used to hide some applications from the main view of the launcher; for example, for data (non-resource) databases, this hides the record count within the launcher info screen.
`eLaunchableData`	When this flag is set, this data database (this flag isn't applicable for applications) can be "launched" by passing its name to its owner application (`'appl'` database with the same creator ID) using the `sysAppLaunchCmdOpenNamedDB` action code.
`eRecyclable`	When this flag is set, the database (resource or record) is recyclable; it will be deleted at the next opportunity—generally the next time the database is closed.
`eBundle`	When this flag is set, the database (resource or record) is associated with the application with the same creator ID. It will be beamed and copied along with the application.
`eOpenDB`	When this flag is set, indicates that the database is currently opened.

NOTE: Do not pass this flag when creating a database. It is for system use only!

`eReadOnly`	When this flag is set, indicates that the database is a read-only (ROM) database.
`eResetAfterInstall`	When this flag is set, indicates that the handheld needs to be reset after the database is installed (actually, after the synchronization involving the database is complete).
`eResource`	When this flag is set, indicates that the database is a resource database (as opposed to a record database). If you do not specify this flag, the database is considered a record database. Most conduits work with record databases.

Database Information Retrieval Constants

You can combine the database information retrieval constants together to specify how Sync Manager operations retrieve data about the database.

`SYNC_DB_INFO_OPT_GET_ATTRIBUTES`	Set this flag to indicate that database search operations are to retrieve database attribute information. You can omit this to optimize performance.
`SYNC_DB_INFO_OPT_GET_SIZE`	Set this flag to indicate that database search operations are to retrieve record count and data size information. You can omit this to optimize performance.

Database Open Mode (eDbOpenModes) Constants

You can combine the database open mode constants together to specify in what mode to open a database.

```
enum eDbOpenModes { eDbShowSecret   = 0x0010,
                    eDbExclusive    = 0x0020,
                    eDbWrite        = 0x0040,
                    eDbRead         = 0x0080
                  };
```

The following rules explain how to use the database open mode constants:

- You generally include the `eDbShowSecret` flag; otherwise, some of the Sync Manager functions will not return records that are marked as private.
- If you are opening a database for reading only, specify `(eDbRead | eDbShowSecret)`
- If you are opening a database for reading and writing, specify `(eDbRead | eDbWrite | eDbShowSecret)`
- You can use the `eDbExclusive` flag to open the database in exclusive mode, which means that nothing else on the handheld can use the database until you close it. This also means that your open call will fail if anything else on the handheld is using the database. Note that `eDbExclusive` is of limited value, since applications are not allowed to run on the handheld during synchronization operations.

`eDbShowSecret`	Open the database with full access to the user's secret records. NOTE: only two of the Sync Manager functions are currently affected by this mode: the SyncReadNextRecInCategory and SyncReadNextModifiedRecInCategory functions skip secret records if you open the database without specifying the `eDbShowSecret` flag.

`eDbExclusive`	Open the database for exclusive access. This means that if the database is already opened, you will be denied access. If you successfully open the database in exclusive mode, no one else will be able to access it until you close the database.
`eDbWrite`	Open the database for write access.
`eDbRead`	Open the database for read access.

Miscellaneous Database Flag (eMiscDbListFlags) Constants

The miscellaneous database list flag constants are returned in the `m_miscFlags` of an object of the The CDbList Class structure when you call certain of the Sync Manager functions; for example, the SyncReadDBList function.

```
enum eMiscDbListFlags {
  eMiscDbFlagExcludeFromSync = 0x0080,
  eMiscDbFlagRamBased        = 0x0040
}

#define eExcludeFromSync
                          eMiscDbFlagExcludeFromSync
```

`eMiscDbFlagExcludeFromSync`

When set, indicates that the database is to be excluded from the synchronization operations. This is typically the result of the user disabling synchronization for the application that owns the database on the handheld. Available with Palm OS versions 2.0 or later.

This constant replaces the older constant `eExcludeFromSync`.

`eMiscDbFlagRamBased`

When set, indicates that the database is located in RAM; if not set, the database is stored in ROM. Available with Palm OS versions 3.0 or later.

Option Flag Constants for SyncCloseDBEx

You can combine the database close constants together to specify actions to take when closing a database.

`SYNC_CLOSE_DB_OPT_UPDATE_BACKUP_DATE`	Specify this to indicate that the database's backup date is to be updated after it is closed.
`SYNC_CLOSE_DB_OPT_UPDATE_MOD_DATE`	Specify this to indicate that the database's modification date is to be updated after it is closed.

Record Attributes (eSyncRecAttrs) Constants

The record attribute constants are combined together in the `m_Attribs` field of The CRawRecordInfo Class.

```
enum eSyncRecAttrs {eRecAttrDeleted = 0x80,
                    eRecAttrDirty   = 0x40,
                    eRecAttrBusy    = 0x20
                    eRecAttrSecret = 0x10,
                    eRecAttrArchived  = 0x08
};
```

`eRecAttrDeleted`	Indicates that the record has been marked as deleted. This replaces the older constant `DELETE_BIT`.
`eRecAttrDirty`	Indicates that the record has been marked as dirty (modified). This replaces the older constant `DIRTY_BIT`.
`eRecAttrSecret`	Indicates that the record has been marked as private. This replaces the older constant `PRIVATE_BIT`.
`eRecAttrBusy`	This value is for system use only. Do not set this bit.

`eRecAttrArchived`
: Indicates that the record has been marked for archiving. This replaces the older constant `ARCHIVE_BIT`.

Search Option Constants

You can use these constants as values in the `srchflags` field of the `SyncFindDbByTypeCreatorParams` structure.

`SYNC_DB_SRCH_OPT_NEW_SEARCH`
: Specify this in the first call to the function with new search criteria to indicate that a new search is to be started. You need to clear this flag before making subsequent calls with the same criteria.

`SYNC_DB_SRCH_OPT_ONLY_LATEST`
: Specify this to indicate that the search should look for the latest version.

Sync Manager Classes

This section describes the classes that you use with the Sync Manager functions. Sync Manager functions use objects of these classes as parameter values.

The CCallModuleParams Class

The SyncCallRemoteModule function uses an object of the `CCallModuleParams` class to specify information that is sent to a module on the handheld.

```
class CCallModuleParams
{
public:
    // Values passed in by the caller:
  DWORD         m_dwCreatorID;
  DWORD         m_dwTypeID;
  WORD          m_wActionCode;
  DWORD         m_dwParamSize;
  void*         m_pParam;
```

```
    DWORD        m_dwResultBufSize;
    void*        m_pResultBuf;

      // Values returned to the caller:
    DWORD        m_dwResultCode;

    DWORD        m_dwActResultSize;
    DWORD        m_dwReserved;
};
```

CCallModuleParams Class Members

--> `m_dwCreatorID`	The creator ID of the application that you are calling on the handheld.
--> `m_dwTypeID`	The type ID of the application that you are calling on the handheld.
--> `m_wActionCode`	The application-specific action code.
--> `m_dwParamSize`	The size of the parameter block.
--> `m_pParam`	A pointer to the actual parameter block.
--> `m_dwResultBufSize`	The size of the results buffer.
<-- `m_pResultBuf`	A pointer to the results buffer.
<-- `m_dwResultCode`	The result code returned by handheld module
<-- `m_dwActResultSize`	The actual size of the result data. This value will be greater than the `m_dwResultBufSize` value if your buffer was not large enough to accommodate all of the results data. In this case, only `dwResultBufSize` bytes of data are copied into the buffer.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The CCardInfo Class

Objects of the `CCardInfo` class specify information about a memory card on the handheld.

```
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[SYNC_REMOTE_CARDNAME_BUF_SIZE];
  char
m_ManufName[SYNC_REMOTE_MANUFNAME_BUF_SIZE];
  WORD     m_romDbCount;
  WORD     m_ramDbCount;
  DWORD    m_dwReserved;
};
```

CCardInfo Class Members

<-- `m_CardNo`	The card number.
<-- `m_CardVersion`	The version of the card.
<-- `m_CreateDate`	The creation date of the card. This is a `t_time` value.
<-- `m_RomSize`	The amount of ROM on the card.
<-- `m_RamSize`	The amount of RAM on the card.
<-- `m_FreeRam`	The amount of available RAM on the card.
<-- `m_CardNameLen`	The number of characters in the card name.

<-- `m_ManufNameLen`	The number of character in the manufacturer's name.
<-- `m_CardName`	The card name string.
<-- `m_ManufName`	The manufacturer's name string.
<-- `m_romDbCount`	The number of ROM-based databases on the card.
<-- `m_ramDBCount`	The number of RAM-based databases on the card.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The CDbCreateDB Class

You use objects of the `CDbCreateDB` class to specify information about a database that you are creating with the `SyncCreateDB` function.

```
class CDbCreateDB
{
public:
  BYTE          m_FileHandle;
  DWORD         m_Creator;
  eDbFlags      m_Flags;
  BYTE          m_CardNo;
  char          m_Name[SYNC_DB_NAMELEN];
  DWORD         m_Type;
  WORD          m_Version;
  DWORD         m_dwReserved;
};
```

CDbCreateDB Class Members

<-- `m_FileHandle`	Upon return, the handle to the database that was created.
--> `m_Creator`	The creator ID for the database. Note that the creator ID should be registered on the Palm OS web site before an application is released. Palm reserves values consisting of all lowercase characters.
--> `m_Flags`	The creation flags for the database. You can specify values from the Database Flag (eDbFlags) Constants.
--> `m_CardNo`	The number of the memory card on which the database is to be stored. Current implementations only support one memory card, the number of which is `0`.
--> `m_Name`	The name of the new database.
--> `m_Type`	The database type. Specify a 4-byte type identifier value for the database. Palm associates special behavior with several identifier value, including `'DATA'`, `'data'`, `'appl'`, `'panl'`, and `'libr'`.
--> `m_Version`	The database version.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The CDbGenInfo Class

You use objects of the `CDbGenInfo` class to specify information about the application information or sorting information blocks in the database file header. You use objects of this class with the SyncReadDBAppInfoBlock, SyncReadDBSortInfoBlock, SyncWriteDBAppInfoBlock, and SyncWriteDBSortInfoBlock functions.

```
class CDbGenInfo
{
public:
  char       m_FileName[SYNC_DB_NAMELEN];
  WORD       m_TotalBytes;
  WORD       m_BytesRead;
  BYTE*      m_pBytes;
  DWORD      m_dwReserved;
};
```

CDbGenInfo Class Members

--> `m_FileName`	The name of handheld database file. Note: this field is not used with the SyncReadDBAppInfoBlock, SyncReadDBSortInfoBlock, SyncWriteDBAppInfoBlock, or SyncWriteDBSortInfoBlock functions.
--> `m_TotalBytes`	When reading an information block, this is the size of the buffer pointed to by `m_pBytes`. When writing an information block, you must set both `m_TotalBytes` and `m_BytesRead` to the size of the block that you are writing (sending).
<-> `m_BytesRead`	When reading an information block, this is the actual size of the block. If the block is larger than the size of your buffer, the behavior depends on which version of the Sync Manager API you are using, as follows: if you are using version 2.1 or later of the Sync Manager API, the first `m_TotalBytes` of block data is copied to your buffer, and the `m_BytesRead` field of `rInfo` is set to the actual block size.

if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the `m_BytesRead` field of `rInfo` is set to the actual block size.

When writing an information block, you must set both `m_TotalBytes` and `m_BytesRead` to the size of the block that you are writing (sending).

--> `m_pBytes`	A pointer to the buffer that you have allocated for the information block data.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The CDbList Class

Each object of the `CDbList` class contains information about a database on the handheld. Objects of this class are used with data structures and functions, including the SyncReadDBList function.

```
class CDbList
{
public:
  int        m_CardNum;
  WORD       m_DbFlags
  DWORD      m_DbType;
  char       m_Name[SYNC_DB_NAMELEN];
  DWORD      m_Creator;
  WORD       m_Version;
  DWORD      m_ModNumber;
  WORD       m_Index;
  long       m_CreateDate;
  long       m_ModDate;
  long       m_BackupDate;
  __int32    m_miscFlags;
  long       m_RecCount;
  long       m_dwReserved;
};
```

CDBList Class Members

<-- `m_CardNum`	The number of the memory card on which the database is stored.
<-- `m_DbFlags`	A combination of the database flags, as described in Database Flag (eDbFlags) Constants.
<-- `m_DbType`	Upon return, the database type, which is a 4-byte type identifier value for the database.
<-- `m_Name`	The name of the database.
<-- `m_Creator`	The creator ID for the database.
<-- `m_Version`	The database version.
<-- `m_ModNumber`	The database modification number.
<-- `m_Index`	The index of the database in the list of databases on the handheld. **Note:** this value is not set for the `SyncReadOpenDbInfo`, `SyncFindDbByName`, or `SyncFindDbByTypeCreator` functions.
<-- `m_CreateDate`	The date on which the database was created. This is a `t_time` value.
<-- `m_ModDate`	The date of the most recent database modification. Note: versions 1.x of the Palm OS did not update the modification date. This is a `t_time` value.
<-- `m_BackupDate`	The date of the most recent database backup. This is a `t_time` value.
<-- `m_miscFlags`	Miscellaneous flags for the database. You can combine values from the miscellaneous database constants, as described in Miscellaneous Database Flag (eMiscDbListFlags) Constants.
--- `m_RecCount`	Currently unused.

--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The CPositionInfo Class

You use objects of the `CPositionInfo` class to specify parameter information for the SyncReadPositionXMap function.

```
class CPositionInfo
{
public:
  BYTE   m_FileHandle;
  WORD   m_FirstPos;
  WORD   m_MaxEntries;
  WORD   m_NumReadIn;
  WORD   m_TotalBytes;
  BYTE * m_pBytes;
};
```

CPositionInfo Class Members

--> `m_FileHandle`	The handle of the database.
--> `m_FirstPos`	The index of the first record ID to read. This index is zero based.
--> `m_MaxEntries`	The total number of record IDs to read. Note that this value must be consistent with the value of the `m_TotalBytes` field: since each record ID is four bytes long, the value of `m_totalBytes` must equal `n_MaxEntries * 4`.
<-- `m_NumReadIn`	Upon return, the index of the record immediately following the last record whose ID was retrieved. If you specify an out-of-bounds value for `m_FirstPos`, then `n_NumReadIn` is set to `0`. Note that the name of this field is very misleading.

--> `m_TotalBytes`	The length, in bytes, of the buffer pointed to by `m_pBytes`. Note that this value must be consistent with the value of the `m_MaxEntries` field: since each record ID is four bytes long, the value of `m_totalBytes` must equal `n_MaxEntries * 4`.
--> `m_pBytes`	The data buffer, which you must allocate before calling the `SyncReadPositionXMap` function.

The CRawPreferenceInfo Class

Use objects of the `CRawPreferenceInfo` class to read or write an application preference. Objects of this class are used with the SyncReadAppPreference and SyncWriteAppPreference functions.

```
class CRawPreferenceInfo
{
public:
  WORD     m_version;
  DWORD    m_creator;
  WORD     m_prefId;
  WORD     m_reqBytes;
  WORD     m_retBytes;
  WORD     m_actSize;
  BOOL     m_backedUp;
  long     m_nBytes;
  BYTE*    m_pBytes;
  DWORD    m_dwReserved;
};
```

CRawPreferenceInfo Class Members

<-> `m_version`	The preference version. If you are writing preference information, fill this in. If you are reading preference information, this is filled in upon return.
--> `m_creator`	The preference creator ID.
--> `m_prefId`	The preference ID.

--> `m_reqBytes`	This field is not used when writing a preference. If you are reading a preference, fill this in with the number of preference bytes that you are requesting. Specify a value of `0xFFFF` to retrieve all preference bytes.
<-- `m_retBytes`	This field is not used when writing a preference. Upon return from reading a preference, this is the number of preference bytes that were copied to the buffer.
<-- `m_actSize`	This field is not used when writing a preference. Upon return from reading a preference, this is the actual size of the preference information on the handheld.
--> `m_backedUp`	This field is used for both reading and writing of preferences. Specifies whether the database for the preference is the saved preferences database of the unsaved preferences database.
--> `m_nBytes`	When reading a preference, this is the size, in bytes, of the buffer pointed to by `m_pBytes`. When writing a preference, this is the number of bytes of preference information to be written.
--> `m_pBytes`	The data buffer, which you must allocate before calling a function with this object.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The CRawRecordInfo Class

You use objects of the CRawRecordInfo class with many of the Sync Manager functions to exchange database records and resources with the handheld. Each of the functions require that you fill in certain of the object fields with information before calling the function; which fields must be filled in are indicated in the documentation for each function.

```
class CRawRecordInfo
{
public:
  BYTE     m_FileHandle;
  DWORD    m_RecId;
  WORD     m_RecIndex;
  BYTE     m_Attribs;
  short    m_CatId;
  int      m_ConduitId;
  DWORD    m_RecSize;
  WORD     m_TotalBytes;
  BYTE*    m_pBytes;
  DWORD    m_dwReserved;
};
```

CRawRecordInfo Class Members

--> `m_FileHandle`	The handle to the database on the handheld. This is a handle returned by the SyncCreateDB or SyncOpenDB functions.
<-> `m_RecId`	The record ID for a record database, and the resource type for a resource database. You supply this when calling a function to read or delete a record by ID. When you call a function to read a record by index (or iteration), this is filled in by the handheld.
<-> `m_RecIndex`	The record index. You supply this when calling a function to read records or resources by index value. If you are deleting a resource, you assign the resource ID to this field. If you are reading a resource, the handheld fills this in with the resource ID. If you are reading a record using version 2.1 or later of the Sync Manager API, this is filled in with the record index by the handheld.

<-> `m_Attribs`	The record attributes. You supply this when calling a function to write a record. When you call a function to read a record, this is filled in by the handheld. This is a combination of values described in Record Attributes (eSyncRecAttrs) Constants.
<-> `m_CatId`	The record's category index. You supply this when calling a function to write a record. When you call a function to read a record, this is filled in by the handheld.
--- `m_ConduitId`	Currently unused.
--> `m_RecSize`	The actual amount of data in the record or resource. If you are calling a function to write a record, this is the record data size and should have the same value as the `m_TotalBytes` field. If you are calling a function to read a record, this is the actual record or resource size. If the record size is larger than the buffer you allocated, this value will be larger than the `m_TotalBytes` value, and the behavior is version dependent: for version 2.1 or later of the Sync Manager API, only `m_TotalBytes` of the record data are copied. for versions of the Sync Manager API earlier than 2.1, nothing is copied.
--> `m_TotalBytes`	The size of the buffer that you have allocated for data. If you are calling a function to write a record or resource, this is the record size; if you are calling a function to read a record or resource, this is the size of the buffer that is pointed to by `m_pBytes`. NOTE: the buffer size is limited to 64K bytes.

--> `m_pBytes`	A pointer to a data buffer that you must allocate prior to passing this object to a function that reads or writes a record.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The CSystemInfo Class

You can use objects of the `CSystemInfo` class to retrieve information about the handheld. This class is used with the SyncReadSystemInfo function.

```
class CSystemInfo
{
public:
  DWORD   m_RomSoftVersion;
  DWORD   m_LocalId;
  BYTE    m_ProdIdLength;
  BYTE    m_AllocedLen;
  BYTE*   m_ProductIdText;
  DWORD   m_dwReserved;
};
```

CSystemInfo Class Members

<-- `m_RomSoftVersion`	Upon return, the software version of the ROM in the handheld. NOTE: you can use the `SYNCROMVMAJOR(val)` and `SYNCROMVMINOR(val)` macros to decode this value into the major and minor version number values.
--- `m_LocalId`	Upon return, the localization ID. This is currently unused, and is always set to `0`.
<-- `m_ProdIdLength`	Upon return, the actual length of the product ID that is stored into the `m_ProductIdText` field.

--> `m_AllocedLen`
: The number of bytes that you allocated for the `m_ProductIdText` buffer.

--> `m_ProductIdText`
: A buffer for storing the product ID. You must allocate this buffer before calling a function with this object. This buffer must be at least `SYNC_MAX_PROD_ID_SIZE` bytes long. The SyncReadSystemInfo function stores the handheld information into this buffer. Currently, all handhelds return the same four bytes:

  `0x00, 0x01, 0x00, 0x00`

--> `m_dwReserved`
: Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The CUserIDInfo Class

You can use objects of the CUserIDInfo class to retrieve information about the user on the handheld. You use this class with the SyncReadUserID function.

```
class CUserIDInfo
{
public:
  char  m_pName[SYNC_REMOTE_USERNAME_BUF_SIZE];
  int   m_NameLength;
  char  m_Password
              [SYNC_REMOTE_PASSWORD_BUF_SIZE];
  int   m_PasswdLength;
  long  m_LastSyncDate;
  DWORD m_LastSyncPC;
  DWORD m_Id;
  DWORD m_ViewerId;
  DWORD m_dwReserved;
};
```

CUserIDInfo Class Members

<-- `m_pName`	Upon return, the user name of the handheld.
<-- `m_NameLength`	Upon return, the actual length of the user name stored in the `m_pName` field.
<-- `m_Password`	Upon return, the user's encrypted password, in binary format.
<-- `m_PasswdLength`	Upon return, the actual length of the user's encrypted password.
<-- `m_LastSyncDate`	The date of the most recent synchronization for the handheld. This is a `t_time` value. NOTE: this field is set to 0 when you call the SyncReadUserID function to read information from a handheld running a version of the Palm OS earlier than version 3.0. For versions 3.0 and later of the Palm OS, this value is set correctly.
<-- `m_LastSyncPC`	The HotSync ID of the desktop computer with which the handheld was most recently synchronized. This value is created when the HotSync Manager is installed, and is stored in the registry.
<-- `m_Id`	Upon return, user ID of the handheld.
<-- `m_ViewerId`	Upon return, the ID of the handheld. Not currently used.
--> `m_dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

Sync Manager Data Structures

This section describes the data structures that you use with the Sync Manager functions.

SyncDatabaseInfoType

Usage

The `SyncDatabaseInfoType` structure is passed to several of the Sync Manager functions, which fill in the fields of the structure with information about a database. The `SyncFindDbByName`, `SyncFindDbByTypeCreator`, and `SyncReadOpenDbInfo` functions use this structure type.

Declaration

```
typedef struct SyncDatabaseInfoType {
  CDbList  baseInfo;
  DWORD    dwNumRecords;
  DWORD    dwTotalBytes;
  DWORD    dwDataBytes;
  DWORD    dwAppBlkSize;
  DWORD    dwSortBlkSize;
  DWORD    dwMaxRecSize;
  DWORD    dwLocalId;
  DWORD    dwOpenRef;
  DWORD    dwReserved;
} SyncDatabaseInfoType;
```

Fields

`<- baseInfo`	Basic information about the database list.
`<- dwNumRecords`	The number of records or resources in the database. This field is only filled in if the `SYNC_DB_INFO_OPT_GET_SIZE` flag is set.
`<- dwTotalBytes`	The total bytes of storage used by database, including overhead. This field is only filled in if the `SYNC_DB_INFO_OPT_GET_SIZE` flag is set.

`<- dwDataBytes`	The total bytes of storage used for data. This field is only filled in if the `SYNC_DB_INFO_OPT_GET_SIZE` flag is set.
`<- dwAppBlkSize`	The block size, in bytes, of the application information block. This field is only filled in for the `SyncReadOpenDbInfo` call when the `SYNC_DB_INFO_OPT_GET_SIZE` option is set.
`<- dwSortBlkSize`	The block size, in bytes, of the sort information block. This field is only filled in for the `SyncReadOpenDbInfo` call when the `SYNC_DB_INFO_OPT_GET_SIZE` option is set.
`<- dwMaxRecSize`	The size of the largest record or resource in the database. This field is only filled in for the `SyncReadOpenDbInfo` call when the `SYNC_DB_INFO_OPT_GET_MAX_REC_SIZE` options is set.
`<- dwLocalID`	For internal use only.
`<- dwOpenRef`	For internal use only.
`-> dwReserved`	Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

The SyncProdCompInfoType Structure

Usage

The `SyncProdCompInfoType` structure is used to retrieve product compatibility information from the handheld. A pointer to a structure of this type is passed as a parameter to the SyncReadProdCompInfo function.

Declaration

```
typedef struct SyncProdCompInfoType {
  SyncVersionType   dlpVer;
  SyncVersionType   compVer;
  DWORD             dwReserved1;
  DWORD             dwReserved2;
  } SyncProdCompInfoType;
```

Fields

<-- `dlpVer`	The version of the desktop link protocol that is installed on the handheld.
<-- `compVer`	Product compatibility version information for the handheld.
--> `dwReserved1`	Reserved for future use. You must set this field to `0` before using this structure.
--> `dwReserved2`	Reserved for future use. You must set this field to `0` before using this structure.

SyncFindDbByNameParams

Usage The `SyncFindDbByNameParams` structure is used to specify information used for finding a database with the `SyncFindDbByName` function.

Declaration

```
typedef struct SyncFindDbByNameParams {
  BYTE    bOptFlags;
  DWORD   dwCardNum;
  char*   pcDatabaseName;
} SyncFindDbByNameParams;
```

Fields

-> `bOptFlags`	The option flags for the find. You can combine values from the Database Information Retrieval Constants.
-> `dwCardNum`	The number of the memory card on which the database resides. The first card in the system is card number `0`, and subsequent card numbers are incremented by `1`.
-> `pcDatabaseName`	A null-terminated string that specifies the database name.

SyncFindDbByTypeCreatorParams

Usage The `SyncFindDbByTypeCreatorParams` structure is used to specify information used for finding a database with the `SyncFindDbByTypeCreator` function.

Declaration

```
typedef struct SyncFindDbByTypeCreatorParams {
  BYTE    bOptFlags;
  BYTE    bSrchFlags;
  DWORD   dwType;
  DWORD   dwCreator;
} SyncFindDbByTypeCreatorParams;
```

Fields

-> `bOptFlags`	The option flags for the find. You can combine values from the Database Information Retrieval Constants.
-> `bSrchFlags`	The search options for finding the database. You can combine values from the Search Option Constants.
-> `dwType`	The database type. Specify a 4-byte type identifier value for the database. You can specify a value of `0` to perform a wildcard search for any database type.
-> `dwCreator`	The database creator ID. Specify a value of `0` to search for any database creator ID.

SyncReadOpenDbInfoParams

Usage The `SyncReadOpenDbInfoParams` structure is used to specify information for retrieving handheld database information with the `SyncReadOpenDbInfo` function.

Declaration

```
typedef struct SyncReadOpenDbInfoParams {
  BYTE    bOptFlags;
  BYTE    bDbHandle;
} SyncReadOpenDbInfoParams;
```

Fields

-> `bOptFlags`	The option flags for the open. You can combine values from the Database Information Retrieval Constants.
-> `bDbHandle`	A handle to an open database, as returned from a call to `SyncOpenDB` or `SyncCreateDB`.

The SyncVersionType Structure

Usage The `SyncVersionType` structure is used to specify a version number.

Declaration

```
typedef struct SyncVersionType {
  WORD   wMajor;
  WORD   wMinor;
} SyncVersionType;
```

Fields

--> `wMajor`	The major protocol number. If the value of this field is `0`, the major protocol number is not available.
--> `wMinor`	The minor protocol number.

Sync Manager Function Summary

You can use the following Sync Manager functions in your conduits:

- SyncAddLogEntry
- SyncCallRemoteModule
- SyncChangeCategory
- SyncCloseDB
- SyncCloseDBEx
- SyncCreateDB
- SyncDeleteAllResourceRec
- SyncDeleteDB
- SyncDeleteRec
- SyncDeleteResourceRec
- SyncFindDbByName
- SyncFindDbByTypeCreator
- SyncGetAPIVersion
- SyncGetDBRecordCount
- SyncGetHHOSVersion

- SyncHHToHostDWord
- SyncHHToHostWord
- SyncHostToHHDWord
- SyncHostToHHWord
- SyncMaxRemoteRecSize
- SyncOpenDB
- SyncPurgeAllRecs
- SyncPurgeAllRecsInCategory
- SyncPurgeDeletedRecs
- SyncReadAppPreference
- SyncReadDBAppInfoBlock
- SyncReadDBList
- SyncReadDBSortInfoBlock
- SyncReadFeature
- SyncReadNextModifiedRec
- SyncReadNextModifiedRecInCategory
- SyncReadNextRecInCategory
- SyncReadOpenDbInfo
- SyncReadPositionXMap
- SyncReadProdCompInfo
- SyncReadRecordById
- SyncReadRecordByIndex
- SyncReadResRecordByIndex
- SyncReadSingleCardInfo
- SyncReadSysDateTime
- SyncReadSystemInfo
- SyncReadUserID
- SyncRebootSystem
- SyncRegisterConduit
- SyncResetRecordIndex

- SyncResetSyncFlags
- SyncUnRegisterConduit
- SyncWriteAppPreference
- SyncWriteDBAppInfoBlock
- SyncWriteDBSortInfoBlock
- SyncWriteRec
- SyncWriteResourceRec
- SyncWriteSysDateTime
- SyncYieldCycles

NOTE: You cannot use the Conduit Development Kit to develop conduits for Hot Sync Manager versions 1.0 or 1.2. For information about versions of the HotSync Manager application and Sync Manager API versions, see HotSync Manager and Sync Manager API Versions.

SyncAddLogEntry

Purpose Adds an entry to the log on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncAddLogEntry(const char* pText);`

Parameters

--> `pText`	The null-terminated log entry string.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_LIMIT_EXCEEDED`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use this function to add an entry to the message log on the handheld. Since the log has limited space, keep your entries as short as possible.

To include a new line in your log entry, use a single line-feed character (character code `0x0A`).

Note that the HotSync Manager application automatically logs the general success or failure status of your conduit; thus, you need not add an entry for this purpose.

SyncCallRemoteModule

Purpose Calls a module (an application, panel, or other executable) on the handheld and returns data and status information to your conduit from that module.

Note that almost all conduits can accomplish their tasks without needing to use this function, which is provided as a "back door" function.

Compatibility

Palm OS version	Sync Manager version
2.0 or later	2.1 or later

Prototype

```
long SyncCallRemoteModule(CCallModuleParams*
pParams);
```

Parameters

<-> `pParams` An object of The CCallModuleParams Class, which contains information for the called module, and returns information to you from the called module, in the following fields:

--> `m_dwCreatorId`
The creator ID of the target application.

--> `m_dwTypeID`
The type ID of the target application.

--> `m_wActionCode`
The action code selector. This value is specific to the module that you are calling.

--> `m_dwParamSize`
The parameter block size.

--> `m_pParam`
A pointer to the parameter block.

--> `m_dwResultBufSize`
The size of the results buffer.

<-> `m_pResultBuf`
A pointer to the results buffer.

<-- `m_dwResultCode`
The result code returned by the handheld module.

<-- `m_dwActResultSize`
The actual data size returned by the handheld module.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_UNKNOWN_REQUEST`
`SYNCERR_LOCAL_BUFF_TOO_SMALL`

The `SYNCERR_UNKNOWN_REQUEST` error is returned if the handheld module was not found, or if the handheld module did not handle the action code.

The `SYNC_LOCAL_BUFF_TOO_SMALL` error is returned if your results buffer was not large enough to contain the results data. If this is the case, then upon return the value of `m_dwActResultSize` will be greater than the value of `m_dwResultBufSize`, and only `m_dwResultBufSize` bytes were copied to the results buffer.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncCallRemoteModule` function to call an application on the handheld while your conduit is running. You can use the parameter block to send arbitrary data to the application. The application can store variable-sized information into the parameter block, which you can examine when the call completes.

Note that the format of the data and the action codes are completely module-specific. The handheld module that you call must have the same structure as a Palm OS application; however, the module can have a proprietary type ID so that it does not show up in the launcher.

Palm discourages you from using this function unless you absolutely have to: almost all conduits can accomplish their jobs without using the `SyncCallRemoteModule` function.

SyncChangeCategory

Purpose Changes the category index of all records in a specified category in an open database on the handheld. This function does not alter the modified status of the records.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

```
long SyncChangeCategory(BYTE fHandle,
BYTE fromIndex, BYTE toIndex);
```

Parameters

--> `fHandle`	A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions. The database must be opened for reading and writing.
--> `fromIndex`	The index of the old category to be changed. This must be a value between `0` and `15`.
--> `toIndex`	The index of the new category. This must be a value between `0` and `15`.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`
`SYNCERR_BAD_OPERATION`
`SYNCERR_READ_ONLY`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncChangeCategory` function to change the category of records in a database on the handheld. All records that are in the category specified by the `fromIndex` parameter value are changed to be in the category specified by the `toIndex` parameter value.

Both category index values must be in the range `0` to `15`. By convention, index `0` is for the unfiled category, and index values `1` through `15` are filed category index values.

SyncCloseDB

Purpose Closes a record or resource database that was opened by SyncOpenDB or SyncCreateDB.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncCloseDB(BYTE fHandle)`

Parameters	--> `fHandle`	A handle to the database on the handheld that was returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions.

Result If successful, returns `0`, which means that the database was closed and its handle was destroyed.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncCloseDB` function to close a database on the handheld that was previously opened by a call to `SyncOpenDB` or `SyncCreateDB`.

The Sync Manager allows only one database to be open at any time; thus, you must close any opened database before calling this function. If you open a database, you must close it before exiting your conduit; otherwise, other conduits will not be able to open their databases.

See Also The SyncCreateDB and SyncOpenDB functions.

SyncCloseDBEx

Purpose

Closes a database, optionally updating its backup and/or modification date. This is the extended version of the `SyncCloseDB` function.

Compatibility

Sync Manager version: 2.2 or later.
Palm OS version: All*.

*This function is compatible with Palm OS version 3.0 if the value of the `bOptFlags` parameter is nonzero.

Prototype

```
long SyncCloseDBEx (BYTE dbHandle,
BYTE bOptFlags);
```

Parameters

`-> dbHandle`	A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions.
`-> bOptFlags`	Option flags for closing the database. You can combine the values specified in Option Flag Constants for SyncCloseDBEx. **Note:** the `bOptFlags` must be 0 for versions of the Palm OS software earlier than version 3.0. When the value is 0, `SyncCloseDBEx` has the same behavior as `SyncCloseDB`.

Result

If successful, returns `0`, which means that the database was closed and its handle was destroyed.

If unsuccessful, returns one of the following nonzero error code values:

```
SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_NO_FILES_OPEN
```

For more information about the error codes, see “Sync Manager Error Code Summary” on page 151.

Comments You can use the `SyncCloseDBEx` function to close a database on the handheld that was previously opened by a call to `SyncOpenDB` or `SyncCreateDB`. This function optionally updates the database modification and/or backup dates.

If the handheld is using a version of the Palm OS software earlier than version 3.0, you must specify a value of 0 for the `bOptFlags` argument; otherwise, this function fails. For version 3.0, you can use the following flag values:

- Set the `SYNC_CLOSE_DB_OPT_UPDATE_BACKUP_DATE` flag to update the backup date of the database to the current date and time on the handheld, without changing the modification date.
- Set the `SYNC_CLOSE_DB_OPT_UPDATE_MOD_DATE` flag to update the modification date of the database to the current date and time on the handheld.

The Sync Manager allows only one database to be open at any time; therefore you must use this function or `SyncCloseDB` before opening another database or exiting your conduit. Otherwise, other conduits will not be able to open their databases.

SyncCreateDB

Purpose Creates a new record or resource database on the handheld, and opens that database. The database is opened for exclusive read-write access, with private (secret) records shown.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncCreateDB (CDbCreateDB& rDbStats)`

Parameters <-> `rDbStats` Database creation information specified in an object of The CDbCreateDB Class, using the following fields:

<-- `m_FileHandle`
Upon return, the handle of the newly created database.

--> `m_Creator`
The creator ID for the database. This value must be registered with Palm, Inc.

Palm reserves creator IDs that consist of all lowercase characters.

--> `m_Flags`
Database creation flags. Use the flags described in Database Flag (eDbFlags) Constants.

--> `m_CardNo`
The card number on which the new database is to be stored. This value must currently be set to `0`.

--> `m_Name`
The name of the new database. This is a null-terminated string whose maximum length, including the terminator byte, is `SYNC_DB_NAMELEN`.

--> `m_Version`
The database version.

--> `m_Type`
The database type. Specify a 4-byte type identifier value for the database. Palm associates special behavior to certain type identifiers, including `'DATA'`, `'data'`, `'appl'`, `'panl'`, and `'libr'`.

Result If successful, returns `0`, which means that the database was created and its handle stored into the `m_FileHandle` data member of `rDbStats`.

If unsuccessful, returns one of the following non-zero error code values:

SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_FILE_ALREADY_EXIST
SYNCERR_TOO_MANY_OPEN_FILES
SYNCERR_FILE_NOT_OPEN

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncCreateDB` function to create a new database on the handheld. You specify information about the database in the `rDbStats` object that you pass in. Upon return, the `m_fileHandle` field contains the file handle for the newly created database.

Note that `SyncCreateDB` will not overwrite an existing database. If you attempt to create a database with the name of an existing database, `SyncCreateDB` fails with the `SYNCERR_FILE_ALREADY_EXIST` error. If you want to replace an existing database, you need to explicitly delete the old database with SyncDeleteDB and then call `SyncCreateDB` to create the new database.

The Sync Manager allows only one database to be open at any time; thus, you must close any opened database before calling this function. If you open a database, you must close it before exiting your conduit; otherwise, other conduits will not be able to open their databases.

SyncDeleteAllResourceRec

Purpose Deletes all resources from an open resource database on the handheld. The database must be opened for reading and writing.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncDeleteAllResourceRec (BYTE fHandle)`

Parameters

--> `fHandle`	A handle to the resource database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions. The database must be opened for reading and writing.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`
`SYNCERR_BAD_OPERATION`
`SYNCERR_READ_ONLY`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncDeleteAllResourceRec` function to delete all of the resources in a resource database on the handheld.

SyncDeleteDB

Purpose Deletes a database on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

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

Parameters

--> `pName`	The name of the database on the handheld. This is a null-terminated string.
--> `nCardNum`	The number of the card on which the database resides on the handheld. For current handhelds, this value must be `0`.

Result If successful, returns `0`, which means that the database was deleted.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`
`SYNCERR_FILE_NOT_OPEN`
`SYNCERR_FILE_ALREADY_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncDeleteDB` function to delete a named database from the handheld.

Note that you cannot delete an open database; you must close the database first.

SyncDeleteRec

Purpose Deletes the specified record from an open record database on the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Comments section for version-based behavior differences.

Prototype `long SyncDeleteRec (CRawRecordInfo &rInfo)`

Parameters

--> `rInfo` An object of The CRawRecordInfo Class, which contains information about the record and database. For this function, you must fill in the following fields in the object:

--> `m_FileHandle`
The handle to an open record database on the handheld, which must be opened for reading and writing.

--> `m_RecId`
The unique record ID of the record to delete.

--> `m_dwReserved`
Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

Result If successful, returns `0`, which means that the record was deleted.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`
`SYNCERR_BAD_OPERATION`
`SYNCERR_READ_ONLY`
`SYNCERR_NOT_FOUND`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncDeleteRec` function to delete a record from a database on the handheld. You specify the record that you want deleted by creating an object of The CRawRecordInfo Class and filling in the object's `m_FileHandle` and `m_RecId` fields. The record data and its entry in the database's record list are completely deleted.

Note that the HotSync iteration index is not updated when you delete a record on a handheld that is running a version of the Palm OS earlier than version 2.0.

You need to be aware of possible difficulties with modifying a database while iterating through it. For more information, see Modifying a Database While Iterating.

SyncDeleteResourceRec

Purpose Deletes a resource from an open resource database on the handheld. The database must be opened for reading and writing.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncDeleteResourceRecord(CRawRecordInfo rRec)`

Parameters

--> `rRec` An object of The CRawRecordInfo Class, which contains information about the resource record and database. For this function, you must fill in the following fields in the object as follows:

--> `m_FileHandle`
The handle to an open resource database, which must be open for reading and writing.

--> `m_RecId`
The 4-byte resource type.

--> `m_RecIndex`
The 2-byte resource ID.

--> `m_dwReserved`
Reserved for future use. You must set this field to `NULL` (`0`) before calling the function.

Result If successful, returns `0`, which means that the resource was deleted.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`
`SYNCERR_BAD_OPERATION`
`SYNCERR_READ_ONLY`
`SYNCERR_NOT_FOUND`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncDeleteResourceRec` function to delete a resource record from a resource database on the handheld. You specify the record that you want deleted by filling in `m_FileHandle`, `m_RecId`, and `m_RecIndex` fields of an object of The CRawRecordInfo Class.

SyncFindDbByName

Purpose Searches for a database by name and memory card number on the handheld, and returns information about the database if it is found.

Compatibility Sync Manager version: 2.2 or later.
Palm OS version: 3.0 or later.

Prototype

```
long SyncFindDbByName (SyncFindDbByNameParams&
rParam, SyncDatabaseInfoType& rInfo);
```

Parameters

`-> rParam`	A structure of type `SyncFindDbByNameParams` that specifies the name, card number, and options for finding the database.
`<- rInfo`	A structure of type `SyncDatabaseInfoType` that specifies information about the found database.

Result If successful, returns `0`.

If unsuccessful, returns one of the following nonzero error code values:

```
SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_NOT_FOUND
```

For more information about the error codes, see "Sync Manager Error Code Summary" on page 151.

Comments You can use the `SyncFindDbByName` function to retrieve information about a database when you know the name of the database. You fill in the structure pointed to by `rParam` with the database name and retrieval options, and `SyncFindDbByName` returns a structure with information filled in.

SyncFindDbByTypeCreator

Purpose Searches for a database by type and creator on the handheld, and returns information about the database if it is found.

Compatibility Sync Manager version: 2.2 or later.
Palm OS version: 3.0 or later.

Prototype

```
long SyncFindDbByTypeCreator
(SyncFindDbByTypeCreatorParams& rParam,
SyncDatabaseInfoType& rInfo);
```

Parameters	`-> rParam`	A structure of type `SyncFindDbByTypeCreatorParams` that specifies the database creator, type, and options for finding the database.
	`<- rInfo`	A structure of type `SyncDatabaseInfoType` that specifies information about the found database.

Result

If successful, returns `0`.

If unsuccessful, returns one of the following nonzero error code values:

```
SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_NOT_FOUND
```

The `SYNCERR_NOT_FOUND` error is returned when there are no more databases on the handheld that meet the search criteria.

For more information about the error codes, see "Sync Manager Error Code Summary" on page 151.

Comments

You can use the `SyncFindDbByTypeCreator` function to retrieve information about a database when you know the type and creator ID of the database. You fill in the structure pointed to by `rParam` with the database type and creator ID values, and specify the retrieval options, and `SyncFindDbByName` returns a structure with information filled in.

You use this function to enumerate through multiple databases of a particular type and/or creator. To begin a new search for a specific creator/type pair, you must specify the `SYNC_DB_SRCH_OPT_NEW_SEARCH` flag in the `bSrchFlags` field of `rParam`. Subsequent calls in the same sequence must exclude this flag.

Note that `SyncFindDbByTypeCreator` does not support creation or deletion of databases in the middle of enumerating through them.

SyncGetAPIVersion

Purpose Retrieves the version of the Sync Manager API that is installed on the desktop computer.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

```
long SyncGetAPIVersion(DWORD* pdwMajor,
DWORD* pdwMinor);
```

Parameters

<-- `pdwMajor`	The major version number of the API on the desktop computer. Specify `NULL` to ignore this value.
<-- `pdwMinor`	The minor version number of the API on the desktop computer. Specify `NULL` to ignore this value.

Result Returns `0`.

Comments You can use the `SyncGetAPIVersion` to retrieve the version of the Sync Manager API on the desktop computer. You can use this information to determine which of the Sync Manager functions you can use on the desktop computer.

For information about Sync Manager API versions and their relationship to HotSync Manager application versions, see HotSync Manager and Sync Manager API Versions.

SyncGetDBRecordCount

Purpose Retrieves the total record or resource count for an open database on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

```
long SyncGetDBRecordCount(BYTE fHandle,
Word &rNumRecs)
```

Parameters

--> `fHandle`	A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions.
<-- `rNumRecs`	The number of records or resources in the database.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

SyncGetHHOSVersion

Purpose Retrieves the version number of the operating system on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	2.1 or later

Prototype `WORD SyncGetHHOSVersion(WORD* pwRomVMinor);`

Parameters

<-- `pwRomVMinor`	The minor version number of the operating system. You can pass `NULL` to ignore this value.

Result If successful, returns the major version number of the operating system on the handheld.

If the function fails, returns `0`, which generally indicates a lost connection.

Comments You can use the `SyncGetHHOSVersion` function to retrieve the version of the operating system that is in use on the handheld with which the Sync Manager is communicating. You can use this information to determine which functions are available on the handheld.

See Also You can also determine the operating system version numbers by calling the SyncReadSystemInfo function and then using the `SYNCROMVMINOR` and `SYNCROMVMINOR` macros on the `m_RomSoftVersion` field of the `CSystemInfo` structure.

SyncHHToHostDWord

Purpose Returns the desktop computer's representation of the specified 32-bit `DWORD` value from the handheld.

Compatibility

Palm OS version	Sync Manager version
All	2.1 or later

Prototype

```
DWORD SyncHHToHostDWord(DWORD dwValue);
```

Parameters

--> `dwValue`	The `DWORD` value from the handheld.

Result The `DWORD` result of the conversion. This is the representation of the value on the desktop computer.

Comments You can use the `SyncHHToHostDWord` to convert a `DWORD` value from the handheld into the representation used on the desktop computer. This function performs byte swapping as required and returns the converted value as the function result.

See Also The SyncHHToHostWord, SyncHostToHHDWord, and SyncHostToHHWord functions.

SyncHHToHostWord

Purpose Returns the desktop computer's representation of the specified 16-bit `WORD` value from the handheld.

Compatibility

Palm OS version	Sync Manager version
All	2.1 or later

Prototype

```
WORD SyncHHToHostWord(WORD wValue);
```

Parameters --> `wValue` The `WORD` value from the handheld.

Result The `WORD` result of the conversion. This is the representation of the value on the desktop computer.

Comments You can use the `SyncHHToHostWord` to convert a `WORD` value from the handheld into the representation used on the desktop computer. This function performs byte swapping as required and returns the converted value as the function result.

See Also The SyncHostToHHDWord, and SyncHostToHHWord functions.

SyncHostToHHDWord

Purpose Returns the handheld's representation of the specified 32-bit `DWORD` value from the desktop computer.

Compatibility

Palm OS version	Sync Manager version
All	2.1 or later

Prototype `DWORD SyncHostToHHDWord(DWORD dwValue);`

Parameters --> `dwValue` The `DWORD` value from the desktop computer.

Result The `DWORD` result of the conversion. This is the representation of the value on the handheld.

Comments You can use the `SyncHostToHHDWord` to convert a `DWORD` value from the desktop computer into the representation used on the handheld. This function performs byte swapping as required and returns the converted value as the function result.

See Also The SyncHostToHHDWord, SyncHHToHostWord, and SyncHostToHHWord functions.

SyncHostToHHWord

Purpose Returns the handheld's representation of the specified 16-bit `WORD` value from the desktop computer.

Compatibility

Palm OS version	Sync Manager version
All	2.1 or later

Prototype `WORD SyncHostToHHWord(WORD wValue);`

Parameters

--> `wValue`	The `WORD` value from the desktop computer.

Result The `WORD` result of the conversion. This is the representation of the value on the handheld.

Comments You can use the `SyncHostToHHWord` to convert a `WORD` value from the desktop computer into the representation used on the handheld. This function performs byte swapping as required and returns the converted value as the function result.

See Also The SyncHHToHostDWord, SyncHHToHostWord, and SyncHostToHHDWord functions.

SyncMaxRemoteRecSize

Purpose Retrieves the maximum record or resource size supported on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	2.2 or later

Prototype `long SyncMaxRemoteRecSize(DWORD& rdwMaxRecSize);`

Parameters <-- `rdwMaxRecSize`

The maximum record size, in bytes, that can be allocated on the handheld. Note that this is the maximum allowed size; the actual maximum size that can be allocated at a specific time is subject to available memory.

A value of 0 indicates that the maximum record size is unknown.

A value of `0xFFFFFFFF` indicates that there is no size limit, subject to available memory.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncMaxRemoteRecSize` function to determine the maximum size record that you can allocate on the handheld. Upon return, the value of the `rdwMaxRecSize` parameter is the maximum record size, in bytes. If this value is `0`, the maximum record size is unknown. If this value is `0xFFFFFFFF`, you can allocate any record size up to the amount of available memory.

Note that the actual size value is subject to available storage.

The maximum record size supported on Palm OS version 3.0 is 65505 bytes. The maximum record size supported for earlier versions is 64720 bytes.

SyncOpenDB

Purpose Opens an existing record or resource database on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

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

Parameters

--> `pName`	The name of the database to open. This is a null-terminated string.
--> `nCardNum`	The memory card on which the database resides. For current handhelds, this value must always be `0`.
<-- `rHandle`	The returned handle to the database.
--> `openMode`	Flag values that specify how to open the database. You can combine values from the Database Open Mode (eDbOpenModes) Constants to form this value.

Result If successful, returns `0`, which means that the database was opened.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`
`SYNCERR_TOO_MANY_OPEN_FILES`
`SYNCERR_FILE_NOT_OPEN`
`SYNCERR_FILE_ALREADY_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncOpenDB` function to open a database by name on the handheld and return a handle to that database. The Sync Manager allows only one database to be open at any time; thus, you must close any opened database before calling this function.

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

You can use the database open flag values to specify how the database is to be opened. For details, see Database Open Mode (eDbOpenModes) Constants.

SyncPurgeAllRecs

Purpose Deletes all of the records in an open record database on the handheld, regardless of record status.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncPurgeAllRecs(BYTE fHandle)`

Parameters

--> `fHandle`	A handle to an opened record database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions. The database must be open for reading and writing.

Result If successful, returns `0`. Also returns 0 if the database has no records.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_READ_ONLY`
`SYNCERR_BAD_OPERATION`
`SYNCERR_NO_FILES_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

SyncPurgeAllRecsInCategory

Purpose Purges all of the records in the specified category in a record database on the handheld.

Compatibility

Palm OS version	Sync Manager version
2.0 or later	All

Prototype

```
long SyncPurgeAllRecsInCategory(BYTE fHandle,
short category);
```

Parameters

--> `fHandle`	A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions.
--> `category`	The index of the category whose records you want deleted. By convention, use a value of `0` for the unfiled category, or use a value between `1` and `15` to specify a filed category.

Result If successful, returns `0`. Also returns 0 if the database has no records.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`
`SYNCERR_BAD_OPERATION`
`SYNCERR_READ_ONLY`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncPurgeAllRecsInCategory` function to delete all records in the specified category from the specified database on the handheld.

Note that this function does not update the record iteration index.

SyncPurgeDeletedRecs

Purpose Deletes all of the records that are marked as deleted or archived from an open record database on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncPurgeDeletedRecs(BYTE fHandle)`

Parameters

--> `fHandle`	A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions. The database must be open for reading and writing.

Result If successful, returns `0`. Also returns 0 if the database has no records.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`

`SYNCERR_LOST_CONNECTION`

`SYNCERR_REMOTE_SYS`

`SYNCERR_REMOTE_MEM`

`SYNCERR_REMOTE_BAD_ARG`

`SYNCERR_READ_ONLY`

`SYNCERR_BAD_OPERATION`
`SYNCERR_NO_FILES_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

SyncReadAppPreference

Purpose Retrieves an application's preferences block from the handheld.

Compatibility

Palm OS version	Sync Manager version
2.0 or later	All

Prototype

```
long SyncReadAppPreference(CRawPreferenceInfo&
rInfo);
```

Parameters

<-> `rInfo` An object of The CRawPreferenceInfo Class, which contains information about the application preference. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is stored. You must allocate this buffer before calling the function.

--> `m_creator`
The 4-byte creator ID for the preference block. This is typically the application creator ID.

--> `m_prefId`
The ID of the preference value that you want to read.

--> `m_backedUp`
A Boolean value; if this is `TRUE`, the *Saved* preferences database is searched; if this is `FALSE`, the *Unsaved* preferences database is searched.

--> `m_reqBytes`
The maximum number of preference bytes requested. This value must not exceed the value of `m_nBytes`.

--> `m_nBytes`
The number of bytes in the `m_pBytes` data array.

<-- `m_version`
The application-specific version number of the preference.

<-- `m_retBytes`
The number of bytes copied to the buffer.

<-- `m_actSize`
The actual size of the preference on the handheld. This will be larger than `m_retBytes` if your buffer was not large enough.

<-- `m_dwReserved`
Reserved for future use. You must set this to `NULL` (`0`) before calling this function.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_UNKNOWN_REQUEST`
`SYNCERR_NOT_FOUND`

The `SYNCERR_NOT_FOUND` error is returned if the requested preference was not found.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadAppPreference` function to retrieve an application's preferences from the handheld. Applications running on version 2.0 or later of the Palm OS can store their non-volatile preferences in one of two preference databases:

- Preferences stored in the *Saved* preference database are backed up during synchronization operations and are automatically restored when required.
- Preferences stored in the *Unsaved* preference database are never backed up or restored by the HotSync Manager application.

The structure of the data in the preferences block is application dependent. The Sync Manager does not modify this data in any way when retrieving it. This means that multi-byte integer data is stored using big-endian byte ordering, with the most significant byte stored at the lower address in memory. Your conduit is responsible for performing any necessary byte swapping.

You must check the version and size of the preference block to ensure compatibility with the application.

To use this function, allocate the `m_pBytes` buffer in an object of The CRawPreferenceInfo Class and fill in the `m_creator`, `m_prefId`, `m_backedUp`, and `m_nBytes` fields of the object. The `SyncReadAppPreferences` function fills in the `m_version`, `m_retBytes`, and `m_actSize` fields of the object.

See Also The SyncWriteAppPreference function.

SyncReadDBAppInfoBlock

Purpose Retrieves the application info block, if one exists, from an open database on the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype

```
long SyncReadDBAppInfoBlock(BYTE fHandle,
CDbGenInfo &rInfo)
```

Parameters

--> `fHandle`	A handle to an open record or resource database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions.
<-> `rInfo`	An object of The CDbGenInfo Class, which contains information about the record and database. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is stored. You must allocate this buffer before calling the function.

--- `m_FileName`
Unused.

--> `m_TotalBytes`
The number of bytes in the `m_pBytes` data array.

<-- `m_BytesRead`
The actual block size, which might be larger than `m_TotalBytes`, as explained in the **Result** section.

--> `m_dwReserved`
Set to `0`.

Result If successful, returns `0`. Note that `SyncReadDBAppInfoBlock` returns `0` even if the buffer you allocated was too small for the resource, with version-specific details as follows:

- if you are using version 2.1 or later of the Sync Manager API, the first `m_TotalBytes` of block data is copied to your buffer, and the `m_BytesRead` field of `rInfo` is set to the actual block size.
- if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the `m_BytesRead` field of `rInfo` is set to the actual block size.

Since the Sync Manager does not generate an error for this condition, you must test for it upon function return with an error code of `0`: if `m_BytesRead` is greater than `m_TotalBytes`, the buffer was too small.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`
`SYNCERR_NO_FILES_OPEN`

The `SYNCERR_NOT_FOUND` error is returned if the requested block is not available.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadDBAppinfoBlock` function to read the application information block from an open database on the handheld. For more information about application information blocks, see The File Header Information Blocks.

To use this function, allocate the `m_pBytes` buffer in an object of The CDbGenInfo Class and fill in the `m_TotalBytes` fields of the object. The `SyncReadDBAppInfo` function fills in the remaining fields of the object.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadDBSortInfoBlock and SyncWriteDBAppInfoBlock functions.

SyncReadDBList

Purpose Retrieves information about a list of databases on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

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

Parameters

--> `cardNo`	The memory card number on which the database(s) reside. For current handhelds, the value of this parameter must be `0`.
--> `startIx`	The starting index for the list of databases for which you want to retrieve information. Specify a value of `0` for the first database, and increment by `1` for each successive database.
--> `bRam`	A Boolean value. If this is `TRUE`, information is retrieved for databases in RAM; if this is `FALSE`, information is retrieved for databases in ROM.
<-> `pList`	An array of objects, each of The CDbList Class. You must preallocate these objects. Upon return, each object contains information about a database on the handheld.
<-> `rCnt`	On entry, the number of objects that you have allocated in `pList`. Upon return, the actual number of contiguous objects that were filled in by this function.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`

The `SYNCERR_NOT_FOUND` error is returned if there are no more databases to list.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadDBList` function to retrieve information about the databases on the handheld. The retrieved information is stored into an array of objects of The CDbList Class that you preallocate. This function can retrieve information about either RAM-based or ROM-based databases, depending on the value of the `bRam` parameter.

To use this function, allocate an array of `CDbList` objects. For more information, see The CDbList Class.

You specify the number of entries that you have allocated in the array in the `rCnt` parameter. Upon return, this value is updated with the actual number of array entries that were filled in by this function.

Note that the Sync Manager can optimize this transaction if you specify a large number of `CDbList` objects. You can call the SyncReadSingleCardInfo function to determine the number of databases on a card and then allocate that many objects. If you cannot use this strategy, 40 objects is a reasonable intermediate array size.

To iterate through all of the databases in RAM or ROM on the handheld, make a series of calls to this function:

- Set `startxX` to `0` for the first call, and subsequently increment it by the number of entries retrieved by the previous call.
- You must remember to reset `rCnt` to the number of entries in your array before each call.

This function is slow when communicating with a handheld that is running a pre-3.0 version of the Palm OS, since those handhelds return only one entry at a time.

SyncReadDBSortInfoBlock

Purpose Retrieves a sort information block, if one exists, from an open record or resource database on the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype

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

Parameters

--> `fHandle`	A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions.

<-> `rInfo`	An object of The CDbGenInfo Class, which contains information about the database. Use the following fields with this function: --- `m_FileName` Ignored. <-> `m_pBytes` The data buffer in which returned data is stored. You must allocate this buffer before calling the function. --> `m_TotalBytes` The number of bytes in the `m_pBytes` data array.
<-- `m_BytesRead`	The actual block size, which might be larger than `m_TotalBytes`, as explained in the section. `--> m_dwReserved` Set to `0`.

Result If successful, returns `0`. Note that `SyncReadDBSortInfoBlock` returns `0` even if the buffer you allocated was too small for the resource, with version-specific details as follows:

- if you are using version 2.1 or later of the Sync Manager API, the first `m_TotalBytes` of block data is copied to your buffer, and the `m_BytesRead` field of `rInfo` is set to the actual block size.
- if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the `m_BytesRead` field of `rInfo` is set to the actual block size.

Since the Sync Manager does not generate an error for this condition, you must test for it upon function return with an error code of `0`: if `m_BytesRead` is greater than `m_TotalBytes`, the buffer was too small.

If unsuccessful, returns one of the following non-zero error code values:

```
SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_NO_FILES_OPEN
SYNCERR_NOT_FOUND
```

The `SYNCERR_NOT_FOUND` error is returned if requested block is not available.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadDBSortInfoBlock` function to read the sorting block from a database on the handheld. For more information about sorting blocks, see The File Header Information Blocks.

To use this function, allocate the `m_pBytes` buffer in an object of The CDbGenInfo Class and fill in the `m_TotalBytes` field of the object. The `SyncReadDBSortInfoBlock` function fills in the `m_BytesRead` field with the actual size of the data block.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadDBAppInfoBlock and SyncWriteDBSortInfoBlock functions.

SyncReadFeature

Purpose

Retrieves a 32-bit feature value from the Feature Manager on the handheld.

Compatibility

Palm OS version	Sync Manager version
2.0 or later	2.1 or later

Prototype

```
long SyncReadFeature(DWORD dwFtrCreator,
WORD wFtrNum, DWORD* pdwFtrValue);
```

Parameters

--> `dwFtrCreator`	The ID of the feature creator.
--> `wFtrNum`	The feature number.
<-- `pdwFtrValue`	The retrieved value of the specified feature.

Result

If successful, returns `0`, which means that the feature was retrieved.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_UNKNOWN_REQUEST`
`SYNCERR_NOT_FOUND`

The `SYNCERR_NOT_FOUND` error is returned if the requested feature could not be found, which indicates that it is not registered.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadFeature` function to retrieve a feature value that is registered with the Feature Manager on the handheld.

Features are stored in volatile storage that is erased and re-initialized during system reset. The Palm OS and applications can register features using their own creator ID. The contents of features are completely application-specific.

SyncReadNextModifiedRec

Purpose An iterator function that retrieves the next modified, archived, or deleted record from an opened record database on the handheld. Each call retrieves the next modified record until all modified records have been returned.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype

```
long SyncReadNextModifiedRec(CRawRecordInfo
&rInfo)
```

Parameters

<-> `rInfo` An object of The CRawRecordInfo Class, which contains information about the record and database. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is stored. You must allocate this buffer before calling the function.

--> `m_FileHandle`
The handle to the open database.

--> `m_TotalBytes`
The number of bytes that you allocated in the `m_pBytes` buffer.

<-- `m_RecId`
The unique record ID of the record.

<-- `m_Attribs`
Record attributes. This is a combination of the values described in Record Attributes (eSyncRecAttrs) Constants.

<-- `m_CatId`
The record's category index.

<-- `m_RecSize` The actual size of the record, which might be larger than the buffer size, as described in the section.

<-- `m_RecIndex`
If you are using Sync Manager version 2.1 or later, the index of the record, which is zero-based.

This value is undefined for earlier versions of the Sync Manager API.

--> `m_dwReserved`
Set to `0`.

--- `mConduitId`
Ignored.

Result If successful, returns `0`. Note that `SyncReadNextModifiedRec` returns `0` even if the buffer you allocated was too small for the record, with version-specific details as follows:

- if you are using version 2.1 or later of the Sync Manager API, the first `m_TotalBytes` of record data is copied to your buffer, and the `m_RecId`, `m_Attribs`, `m_CatId`, `m_RecIndex`, and `m_RecSize` fields of `rInfo` are filled in correctly.
- if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the

`m_RecId`, `m_Attribs`, `m_CatId`, and `m_RecSize` fields of `rInfo` are filled in correctly.

Since the Sync Manager does not generate an error for this condition, you must test for it upon function return with an error code of `0`: if `m_RecSize` is greater than `m_TotalBytes`, the buffer was too small.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`
`SYNCERR_REMOTE_BUSY`
`SYNCERR_NO_FILES_OPEN`

The `SYNCERR_NOT_FOUND` error is returned if there are no more modified records to retrieve.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadNextModifiedRec` function to retrieve the next modified record from a database on the handheld.

To use this function, allocate the `m_pBytes` buffer in an object of The CRawRecordInfo Class and fill in the `m_TotalBytes` and `m_FileHandle` members of the object. You must fill in `m_TotalBytes` with the size of the buffer that you allocated and assigned to `m_pBytes`.

The `SyncReadNextModifiedRec` retrieves the record and stores it into the buffer. This function also fills in the `m_RecId`, `m_Attribs`, `m_CatId`, `m_RecSize`, and `m_RecIndex` fields in `rInfo`.

For general information about iterating through a handheld database, see Iterating Through a Database.

You need to be aware of possible difficulties with modifying a database while iterating through it. For more information, see Modifying a Database While Iterating.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadNextModifiedRecInCategory, SyncReadNextRecInCategory, and SyncResetRecordIndex functions.

SyncReadNextModifiedRecInCategory

Purpose An iterator function that retrieves modified records, including deleted and archived records, in a category from an open database on the handheld. Each call retrieves the next modified record from the category, until all modified records have been retrieved.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype

```
long SyncReadNextModifiedRecInCategory
(CRawRecordInfo& rInfo);
```

Parameters <-> `rInfo` An object of The CRawRecordInfo Class, which contains information about the record and database. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is

stored. You must allocate this buffer before calling the function.

--> `m_FileHandle`
The handle to an open record database.

--> `m_CatId`
The category index. This must be a value between `0` and `15`.

--> `m_TotalBytes`
The number of bytes that you allocated in the `m_pBytes` buffer.

<-- `m_RecId`
The unique record ID of the record.

<-- `m_Attribs`
Record attributes. This is a combination of the values described in Record Attributes (eSyncRecAttrs) Constants.

<-- `m_RecSize` The actual size of the record, which might be larger than the buffer size, as described in the section.

<-- `m_RecIndex`
If you are using Sync Manager version 2.1 or later, the index of the record, which is zero-based.

This value is undefined for earlier versions of the Sync Manager API.

--> `m_dwReserved`
Set to `0`.

--- `mConduitId`
Ignored.

Result If successful, returns `0`. Note that `SyncReadNextModifiedRecInCategory` returns `0` even if the buffer you allocated was too small for the record, with version-specific details as follows:

- if you are using version 2.1 or later of the Sync Manager API, the first `m_TotalBytes` of record data is copied to your buffer, and the `m_RecId`, `m_Attribs`, `m_RecIndex`, and `m_RecSize` fields of `rInfo` are filled in correctly.
- if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the `m_RecId`, `m_Attribs`, and `m_RecSize` fields of `rInfo` are filled in correctly.

Since the Sync Manager does not generate an error for this condition, you must test for it upon function return with an error code of `0`: if `m_RecSize` is greater than `m_TotalBytes`, the buffer was too small.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`
`SYNCERR_RECORD_BUSY`
`SYNCERR_NO_FILES_OPEN`

The `SYNCERR_NOT_FOUND` error is returned if there are no more modified records to retrieve.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadNextModifiedRecInCategory` function to retrieve the next modified record in the specified category from a database on the handheld.

To use this function, allocate the `m_pBytes` buffer an object of The CRawRecordInfo Class and fill in the `m_CatId`, `m_TotalBytes` and `m_FileHandle` members of the object. You must fill in `m_TotalBytes` with the size of the buffer that you allocated and assigned to `m_pBytes`.

The `SyncReadNextModifiedRecInCategory` function retrieves the record and stores it into the buffer. This function also fills in the `m_RecId`, `m_Attribs`, `m_RecSize`, and `m_RecIndex` fields of `rInfo`.

For general information about iterating through a handheld database, see Iterating Through a Database.

You need to be aware of possible difficulties with modifying a database while iterating through it. For more information, see Modifying a Database While Iterating.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadNextModifiedRec, SyncReadNextRecInCategory, and SyncResetRecordIndex functions.

SyncReadNextRecInCategory

Purpose An iterator function that retrieves any record in a category from an open database on the handheld, including deleted, archived, and modified records. Each call retrieves the next record from the category, until all records have been retrieved.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype

```
long SyncReadNextRecInCategory(CRawRecordInfo&
rInfo);
```

Parameters

<-> `rInfo` An object of The CRawRecordInfo Class, which contains information about the record and database. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is stored. You must allocate this buffer before calling the function.

--> `m_FileHandle`
The handle to the open database.

--> `m_CatId`
The category index.

--> `m_TotalBytes`
The number of bytes that you allocated in the `m_pBytes` buffer.

<-- `m_RecId`
The unique record ID of the record.

<-- `m_Attribs`
Record attributes. This is a combination of the values described in Record Attributes (eSyncRecAttrs) Constants.

<-- `m_RecSize` The actual size of the record, which might be larger than the buffer size, as described in the section.

<-- `m_RecIndex`
If you are using Sync Manager version 2.1 or later, the index of the record, which is zero-based.

This value is undefined for earlier versions of the Sync Manager API.

--> `m_dwReserved`
Set to `0`.

--- `mConduitId`
Ignored.

Result If successful, returns `0`. Note that `SyncReadNextRecInCategory` returns `0` even if the buffer you allocated was too small for the record, with version-specific details as follows:

- if you are using version 2.1 or later of the Sync Manager API, the first `m_TotalBytes` of record data is copied to your buffer, and the `m_RecId`, `m_Attribs`, `m_RecIndex`, and `m_RecSize` fields of `rInfo` are filled in correctly.
- if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the `m_RecId`, `m_Attribs`, and `m_RecSize` fields of `rInfo` are filled in correctly.

Since the Sync Manager does not generate an error for this condition, you must test for it upon function return with an error code of `0`: if `m_RecSize` is greater than `m_TotalBytes`, the buffer was too small.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`
`SYNCERR_RECORD_BUSY`
`SYNCERR_NO_FILES_OPEN`

The `SYNCERR_NOT_FOUND` error is returned if there are no more modified records to retrieve.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadNextRecInCategory` function to retrieve the next record in the specified category from a database on the handheld.

To use this function, allocate the `m_pBytes` buffer in an object of The CRawRecordInfo Class and fill in the `m_CatId`, `m_TotalBytes` and `m_FileHandle` members of the object. You must fill in `m_TotalBytes` with the size of the buffer that you allocated and assigned to `m_pBytes`.

The `SyncReadNextRecInCategory` function retrieves the record and stores it into the buffer. This function also fills in the `m_RecId`, `m_Attribs`, `m_RecSize`, and `m_RecIndex` fields of `rInfo`.

For general information about iterating through a handheld database, see Iterating Through a Database.

You need to be aware of possible difficulties with modifying a database while iterating through it. For more information, see Modifying a Database While Iterating.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadNextModifiedRec, SyncReadNextModifiedRecInCategory, and SyncResetRecordIndex functions.

SyncReadOpenDbInfo

Purpose Retrieves comprehensive information about an open database on the handheld.

Compatibility Sync Manager version: 2.2 or later.
Palm OS version: 3.0 or later.

Prototype

```
long SyncReadOpenDbInfo (SyncReadOpenDbInfoParams&
rParam, SyncDatabaseInfoType& rInfo);
```

Parameters	`-> rParam`	A pointer to a structure of type `SyncReadOpenDbInfoParams` that specifies the database handle and options for opening the database.
	`<- rInfo`	A pointer to a structure of type `SyncDatabaseInfoType` in which the information is returned.

Result If successful, returns `0`.

If unsuccessful, returns one of the following nonzero error code values:

```
SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_NO_FILES_OPEN
```

For more information about the error codes, see "Sync Manager Error Code Summary" on page 151.

Comments You can use the `SyncReadOpenDbInfo` function to retrieve information about a database that is opened on the handheld. To use this function, you need to fill in a `SyncReadOpenDbInfoParams` structure with the database handle and flags that specify how to open the database. The `SyncReadOpenDbInfo` function fills in a `SyncDatabaseInfoType` structure with information about the database.

See Also The `SyncFindDbByName` and `SyncFindDbByTypeCreator` functions.

SyncReadPositionXMap

Purpose Retrieves a list of the record IDs in their sorted order from a database on the handheld. Note that the record ID values are in big-endian (Motorola) byte ordering format.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Comments section for version-based behavior differences.

Prototype `long SyncReadPositionXMap(CPositionInfo& rInfo);`

Parameters

<-> `rInfo` An object of The CPositionInfo Class, which contains the record IDs and other information for the database. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is stored. You must allocate this buffer before calling the function.

--> `m_FileHandle`
The handle to the open record database.

--> `m_FirstPos`
The starting record index. Use `0` for the first record.

--> `m_MaxEntries`
The total number of record IDs to read. Each ID is four bytes long.

--> `m_TotalBytes`
The number of bytes that you allocated in the `m_pBytes` buffer. Be sure to allocate a bufer large enough to hold `m_MaxEntries` record IDs. You can

compute this as `m_MaxEntries * sizeof(DWORD)`.

<-- `m_NumReadIn`
The index of the record immediately following the last record whose ID was retrieved. If you specify an out-of-bounds value for `m_FirstPos`, then `n_NumReadIn` is set to `0`.

Note that the name of this field is very misleading.

Result If successful, returns `0`. Note that `SyncReadPositionXMap` returns `0` if the starting index, `m_FirstPos`, is out of bounds, in which case `m_NumReadIn` is set to `0`.

If unsuccessful, returns one of the following non-zero error code values:

```
SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_BAD_ARG
SYNCERR_NOT_FOUND
SYNCERR_NO_FILES_OPEN
```

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadPositionXMap` function to retrieve a list of record IDs in a handheld database in their sorted order. Some conduits use this function to apply the same ordering to the database on the desktop computer, although it might be less efficient than simply sorting the records.

To use this function, allocate the `m_pBytes` buffer in an object of The CPositionInfo Class and fill in the `m_FirstPos`, `m_MaxEntries`, `m_TotalBytes` and `m_FileHandle` members of the object. You must fill in `m_TotalBytes` with the size of the buffer that you allocated and assigned to `m_pBytes`.

The `SyncReadPositionXMap` function retrieves the record IDs and stores them into the buffer. This function also fills in the `m_NumReadIn` field of `rInfo`.

`SyncReadPositionXMap` does not convert the retrieved record IDs to your desktop computer's byte-ordering convention. Each 4-byte record ID is returned in big-endian (Motorola) byte ordering, with the most significant byte stored in the lower memory address. This differs from the behavior of other Sync Manager record reading functions, which do convert the ID to the desktop computer's byte ordering. This means that when you compare record IDs returned by `SyncReadPositionXMap` with record IDs returned by other Sync Manager functions, you need to first convert the former to the desktop computer's format.

WARNING! There is a bug in the pre-2.2 Sync Manager API versions of the `ReadPositionXMap` function that causes it to crash if you request that it return a subset of the record IDs. This problem is fixed in versions 2.2 and later of the Sync Manager API code.

You can easily get around this problem in earlier versions of the Sync Manager API by having `SyncReadPositionXMap` retrieve all of the record IDs at once. To do so, follow these steps:

1. Retrieve the count of records in the database by calling the SyncGetDBRecordCount function.

2. Allocate the `m_pBytes` buffer to accommodate that many record IDs. Compute the required size as follows:

    bufsize = count * sizeof(DWORD)

3. Call `SyncReadPositionXMap` to retrieve all of the record IDs.

SyncReadProdCompInfo

Purpose Retrieves product compatibility information from the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	

Prototype

```
long SyncReadProdCompInfo(SyncProdCompInfoType&
rInfo);
```

Parameters

`rInfo`	A pointer to a product compatibility information structure, as described in The SyncProdCompInfoType Structure. Upon return, this structure is filled in with compatibility information.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadProdCompInfo` to retrieve product compatibility information from the handheld. The retrieved information includes which version of the desktop link protocol is supported by the handheld.

SyncReadRecordById

Purpose Retrieves a record, by unique record ID, from an open record database on the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype `long SyncReadRecordById(CRawRecordInfo &rInfo)`

Parameters <-> `rInfo` An object of The CRawRecordInfo Class, which contains information about the record and database. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is stored. You must allocate this buffer before calling the function.

--> `m_FileHandle`
The handle to an open record database.

--> `m_RecId`
The record ID.

--> `m_TotalBytes`
The number of bytes that you allocated in the `m_pBytes` buffer.

<-- `m_RecIndex`
If you are using Sync Manager version 2.1 or later, the index of the record, which is zero-based.

This value is undefined for earlier versions of the Sync Manager API.

<-- `m_CatId`
The index of the record's category.

<-- `m_Attribs`
Record attributes. This is a combination of the values described in Record Attributes (eSyncRecAttrs) Constants.

<-- `m_RecSize` The size of the record, which might be larger than the buffer size, as described in the section.

--> `m_dwReserved`
Set to `0`.

--> `m_conduit`
Ignored.

Result If successful, returns `0`. Note that `SyncReadRecordById` returns `0` even if the buffer you allocated was too small for the record, with version-specific details as follows:

- if you are using version 2.1 or later of the Sync Manager API, the first `m_TotalBytes` of record data is copied to your buffer, and the `m_RecId`, `m_Attribs`, `m_CatId`, `m_RecIndex`, and `m_RecSize` fields of `rInfo` are filled in correctly.
- if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the `m_RecId`, `m_Attribs`, `m_CatId`, and `m_RecSize` fields of `rInfo` are filled in correctly.

Since the Sync Manager does not generate an error for this condition, you must test for it upon function return with an error code of `0`: if `m_RecSize` is greater than `m_TotalBytes`, the buffer was too small.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`
`SYNCERR_RECORD_BUSY`
`SYNCERR_NO_FILES_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadRecordById` function to retrieve the specified record, by ID, from a database on the handheld.

To use this function, allocate the `m_pBytes` buffer in an object of The CRawRecordInfo Class, and fill in the `m_RecId`, `m_TotalBytes` and `m_FileHandle` members of the object. You must fill in `m_TotalBytes` with the size of the buffer that you allocated and assigned to `m_pBytes`.

The `SyncReadRecordById` function retrieves the record and stores it into the buffer. This function also fills in the `m_RecIndex`, `m_CatId`, `m_Attribs`, and `m_RecSize` fields of `rInfo`.

You need to be aware of possible difficulties with modifying a database while iterating through it. For more information, see Modifying a Database While Iterating.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadRecordByIndex, SyncReadNextModifiedRec, SyncReadNextModifiedRecInCategory, and SyncReadNextRecInCategory functions.

SyncReadRecordByIndex

Purpose

Retrieves a record, by index, from a record database on the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype

`long SyncReadRecordByIndex(CRawRecordInfo &rInfo)`

Parameters

<-> `rInfo`
An object of The CRawRecordInfo Class, which contains information about the record and database. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is stored. You must allocate this buffer before calling the function.

--> `m_FileHandle`
The handle to an open record database.

--> `m_RecIndex`
The record index. This index is zero-based.

--> `m_TotalBytes`
The number of bytes that you allocated in the `m_pBytes` buffer.

<-- `m_RecId`
The record's unique ID.

<-- `m_CatId`
The record's category.

<-- `m_Attribs`
Record attributes. This is a combination of the values described in Record Attributes (eSyncRecAttrs) Constants.

<-- `m_RecSize` The size of the record, which might be larger than the buffer size, as described in the section.

--> `m_dwReserved`
Set to `0`.

--> `m_conduitId`
Ignored.

Result If successful, returns `0`. Note that `SyncReadRecordByIndex` returns `0` even if the buffer you allocated was too small for the record, with version-specific details as follows:

- if you are using version 2.1 or later of the Sync Manager API, the first m_TotalBytes of record data is copied to your buffer, and the `m_RecId`, `m_Attribs`, `m_CatId`, `m_RecIndex`, and `m_RecSize` fields of `rInfo` are filled in correctly.
- if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the `m_RecId`, `m_Attribs`, `m_CatId`, and `m_RecSize` fields of rInfo are filled in correctly.

Since the Sync Manager does not generate an error for this condition, you must test for it upon function return with an error code of `0`: if `m_RecSize` is greater than `m_TotalBytes`, the buffer was too small.

If unsuccessful, returns one of the following non-zero error code values:

```
SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_NOT_FOUND
SYNCERR_RECORD_BUSY
SYNCERR_NO_FILES_OPEN
```

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadRecordByIndex` function to retrieve the specified record, by index, from a database on the handheld.

To use this function, allocate the `m_pBytes` buffer in an object of The CRawRecordInfo Class and fill in the `m_RecIndex`, `m_TotalBytes` and `m_FileHandle` members of the object. You must fill in `m_TotalBytes` with the size of the buffer that you allocated and assigned to `m_pBytes`.

The `SyncReadRecordByIndex` function retrieves the record and stores it into the buffer. This function also fills in the `m_RecId`, `m_CatId`, `m_Attribs`, and `m_RecSize` fields of `rInfo`.

You need to be aware of possible difficulties with modifying a database while iterating through it. For more information, see Modifying a Database While Iterating.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadRecordById, SyncReadNextModifiedRec, SyncReadNextModifiedRecInCategory, and SyncReadNextRecInCategory functions.

SyncReadResRecordByIndex

Purpose Retrieves a resource record, by index, from an open resource database on the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype

```
long SyncReadResRecordByIndex(CRawRecordInfo
&rInfo, BOOL bBody)
```

Parameters

<-> `rRec` An object of The CRawRecordInfo Class, which contains information about the resource and database. Use the following fields with this function:

<-> `m_pBytes`
The data buffer in which returned data is stored. You must allocate this buffer before calling the function.

--> `m_FileHandle`
The handle to the open resource database.

<-> `m_RecIndex`
On entry, the resource index. Upon return, the resource ID.

--> `m_TotalBytes`
The number of bytes that you allocated in the `m_pBytes` buffer.

<-- `m_RecId`
The resource type.

	<-- `m_CatId` The resource index, which should match the value that you passed in as the value of `m_RecIndex`.
<-- `m_RecSize`	The actual size of the resource, which might be larger than the buffer size, as described in the Result section.
	--> `m_dwReserved` Set to `0`.
	--> `m_conduitId` Ignored.
--> `bBody`	A Boolean value. If this is `TRUE`, the resource data is retrieved and stored into the `m_pBytes` buffer in `rInfo`. If not, the other fields in `rInfo` are filled in, but the resource data is not copied.

Result If successful, returns `0`. Note that `SyncReadResRecordByIndex` returns `0` even if the buffer you allocated was too small for the resource, with version-specific details as follows:

- if you are using version 2.1 or later of the Sync Manager API, the first `m_TotalBytes` of resource data is copied to your buffer, and the `m_RecId`, `m_CatId`, `m_RecIndex`, and `m_RecSize` fields of `rInfo` are filled in correctly.
- if you are using a version of the Sync Manager API earlier than version 2.1, nothing is copied to your buffer, but the `m_RecId`, `m_CatId`, and `m_RecSize` fields of `rInfo` are filled in correctly.

Since the Sync Manager does not generate an error for this condition, you must test for it upon function return with an error code of `0`: if `m_RecSize` is greater than `m_TotalBytes`, the buffer was too small.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NOT_FOUND`
`SYNCERR_NO_FILES_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadResRecordByIndex` function to retrieve the specified resource record, by index, from a resource database on the handheld. This function also fills in the `m_RecId`, `m_RecIndex`, `m_CatId`, and `m_RecSize` fields of `rInfo`.

To use this function, allocate the `m_pBytes` buffer in object of The CRawRecordInfo Class and fill in the `m_RecIndex`, `m_TotalBytes` and `m_FileHandle` members of the object. You must fill in `m_TotalBytes` with the size of the buffer that you allocated and assigned to `m_pBytes`.

To start iterating through a resource database, you can set `m_RecIndex` to 0 and call this function. You can retrieve subsequent resources by incrementing the previous value of `m_RecIndex` by `1`. Continue calling `SyncReadResRecordByIndex` until an error code is returned. You must be sure to “refresh” the value of `m_RecIndex`, because this function overloads that field by returning the resource ID in it.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncWriteResourceRec function.

SyncReadSingleCardInfo

Purpose Retrieves information about a memory card on the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Comments section for version-based behavior differences.

Prototype `long SyncReadSingleCardInfo(CardInfo &rInfo)`

Parameters

<-> `rInfo` An object of The CCardInfo Class. Use the following fields with this function:

--> `m_CardNo`
The number of the card for which you want information. The first card in the system is card number `0`, and subsequent card numbers are incremented by `1`.

<-- `m_CardVersion`
The card format version.

<-- `m_CreateDate`
The card creation date as a `t_time value`. This is `0` or `-1` if the date is not available.

<-- `m_RomSize`
Total ROM size of the card.

<-- `m_RamSize`
Total RAM size of the card, including storage and dynamic heaps.

<-- `m_FreeRam`
Amount of unused RAM on the card; this value is different for different versions of the operating systems, as described in the Comments section.

<-- `m_CardNameLen`
The length of the card name string.

<-- `m_ManufNameLen`
The length of the card manufacturer name string.

<-- `m_CardName`
The card name string. Note that this string is not null-terminated.

<-- `m_ManufName`
The manufacturer name string. Note that this string is not null-terminated.

<-- `m_romDbCount`
The number of ROM-based databases on the card.

<-- `m_ramDbCount`
The number of RAM-based databases on the card.

<-- `m_dwReserved`
Reserved for future use. You must set this to `NULL` (`0`) before calling this function.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_NOT_FOUND

The `SYNCERR_NOT_FOUND` error is returned if the specified card number is outside of the range of available cards.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadSingleCardInfo` function to retrieve information about a memory card that is connected to the handheld. You can use this function to determine the total number of ROM and RAM-based databases prior to calling the SyncReadDBList function.

Currently, all handhelds have exactly one card, which is card number `0`.

The value returned in the `m_FreeRam` field depends on the version of the Palm OS running on the handheld:

- if the handheld is running version 3.0 or later, the value of `m_FreeRam` includes unused RAM in storage heaps only, which is the amount of memory available for records or resources.
- if the handheld is running an earlier version of the Palm OS, the value of `m_FreeRam` is the sum of unused RAM in both storage and dynamic heaps. However, you can only store data in the storage heaps.

See Also The SyncReadDBList function.

SyncReadSysDateTime

Purpose Retrieves the current date and time, according to the system clock on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

```
long SyncReadSysDateTime(long& rDate);
```

Parameters

<-- `rDate`	The system date and time on the handheld, as a `t_time` value. If an error occurs, the value of `rDate` is either `-1` or `0`.

Result

If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments

IMPORTANT: This function is implemented using POSIX time functions in CodeWarrior's MSL library. It's important to note that starting with CodeWarrior Pro 6, Metrowerks changed this function to use an epoch of 1970 instead of 1900. HotSync Manager 3.0 is built using CodeWarrior 7.2, so it's possible your conduit may need to be rebuilt with the new version of MSL to avoid being off by 70 years.

See Also

The SyncWriteSysDateTime function.

SyncReadSystemInfo

Purpose

Retrieves the Palm OS version and product information for the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncReadSystemInfo(CSystemInfo &rInfo)`

Parameters

<-> `rInfo`

An object of The CSystemInfo Class, which contains information about the handheld. For this function, you must first allocate the `m_ProductIdText` buffer and fill in the following fields in the object. Note that you need to allocate `SYNC_MAX_PROD_ID_SIZE` bytes for your buffer.

<-> `m_ProductIdText`
: The data buffer in which returned data is stored. You must allocate this buffer before calling the function. Allocate `SYNC_MAX_PROD_ID_SIZE` bytes for this buffer.

--> `m_AllocedLen`
: The number of bytes that you allocated in the `m_ProductIdText` buffer.

<-- `m_RomSoftVersion`
: The ROM version of the handheld. NOTE: you can use the `SYNCROMVMAJOR(val)` and `SYNCROMVMINOR(val)` macros to decode this value into the major and minor version number values.

<-- `m_LocalId`
: The localization ID for the handheld. Currently, all systems return the value `0x00010000L`.

<-- `m_ProdIdLength`
: The actual number of bytes stored into the `m_ProductIdText` buffer.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadSystemInfo` function to retrieve system information from the handheld.

To use this function, allocate the `m_ProductIdText` buffer in an object of The CSystemInfo Class and fill in the `m_AllocedLen` member of the object. You must fill in `m_AllocedLen` with the size of the buffer that you allocated and assigned to `m_ProductIdText`.

The `SyncReadSystemInfo` function stores the handheld system information into the buffer. The retrieved product ID is a binary (non-ASCII) sequence of bytes. Currently, all handhelds return the same four bytes:

```
0x00, 0x01, 0x00, 0x00
```

This function also fills in the `m_RomSoftVersion`, `m_LocalId`, and `m_ProdIdLength` fields.

See Also The SyncGetHHOSVersion function.

SyncReadUserID

Purpose Retrieves information about the user of the handheld, including the user name, last synchronization date, and encrypted password.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Comments section for version-based behavior differences.

Prototype `long SyncReadUserID(CUserIDInfo &rInfo);`

Parameters

<-- `rInfo`	An object of The CUserIDInfo Class. Upon successful return, the fields of this object are filled with information about the user of the handheld. If the `m_NameLength` field of `rInfo` is `0` upon return, then user information has not yet been established on the handheld.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncReadUserId` function to retrieve information about the user of the handheld. The information is stored into the `CUserIdInfo` object that you pass in as a parameter.

User information is written to a new or reset handheld after completion of a synchronization. If this has not yet occurred on the handheld, the user information is empty, and the `m_NameLength` field of `rInfo` is set to `0`.

WARNING! When you call `SyncReadUserId` to read information from a handheld that is running a version of the Palm OS earlier than version 3.0, the `LastSyncDate` field of the `CUserIdInfo` structure is set to 0. The `LastSyncDate` field is correctly set to the most recent synchronization date for Palm OS 3.0 and later.

See Also The SyncReadSystemInfo function.

SyncRebootSystem

Purpose Sends a request to soft-reset the handheld at the end of synchronization operations.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

```
long SyncRebootSystem(void);
```

Parameters None.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`

For more information about the error codes, see Sync Manager Error Code Summary.

SyncRegisterConduit

Purpose Registers a conduit that is about to start synchronization operations and returns a handle for use with other Sync Manager functions.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncRegisterConduit(CONDHANDLE &rHandle)`

Parameters

<-- `rHandle`	The address of the handle to your conduit.

Result If successful, returns `0`, which means that you can proceed with synchronization.

If the conduit could not be registered, returns `-1`, which means that the previous conduit did not unregister itself and you cannot proceed with synchronization.

If unsuccessful for another reason, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_CANCEL_SYNC`
`SYNCERR_LOCAL_CANCEL_SYNC`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You must call the `SyncRegisterConduit` function to register your conduit before making any other calls into the Sync Manager.

If your call to `SyncRegisterConduit` succeeds, you must call the `SyncUnRegisterConduit` function after you finish synchronizing.

See Also The SyncUnRegisterConduit function.

SyncResetRecordIndex

Purpose Resets the record iteration index of an open record database on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype
```
long SyncResetRecordIndex(BYTE fHandle);
```

Parameters	--> `fHandle`	A handle to an open record database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncResetRecordIndex` to reset the record index for the specified database on the handheld. You call this function before iterating through the database with functions such as `SyncReadNextRecInCategory` or `SyncReadNextModifiedRecInCategory`.

You do not need to call the `SyncResetRecordIndex` function before iterating through a database for the first time; however, if your conduit is iterating through the same database more than once, you need to call this function to reset the index before beginning the second (or subsequent) iteration.

For general information about iterating through a handheld database, see Iterating Through a Database.

You need to be aware of possible difficulties with modifying a database while iterating through it. For more information, see Modifying a Database While Iterating.

See Also The SyncReadNextModifiedRec, SyncReadNextModifiedRecInCategory, and SyncReadNextRecInCategory functions.

SyncResetSyncFlags

Purpose Resets the modified flag of all records in the opened record database on the handheld. Resets the backup date for an opened record or resource database.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

```
long SyncResetSyncFlags(BYTE fHandle)
```

Parameters

--> `fHandle`	A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions. If `fHandle` is a handle to a record database, the database must be opened for reading and writing. If `fHandle` is a handle to a resource database, the database can be opened for either read-only or read-write access.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`
`SYNCERR_READ_ONLY`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the SyncResetSyncFlags function to clear the dirty (modified) flag of all records in the specified database. For more information about the record flags, see Record Attributes (eSyncRecAttrs) Constants.

SyncUnRegisterConduit

Purpose Unregisters a conduit that was successfully registered with a previous call to the SyncRegisterConduit function, and cleans up any system resources that the Sync Manager allocated for the conduit.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncUnRegisterConduit(CONDHANDLE handle)`

Parameters

--> `handle`	A handle to a conduit that was returned from a previous call to the SyncRegisterConduit function.

Result If successful, returns `0`, which means that your conduit was successfully unregistered.

If `handle` is not a valid conduit handle, returns `-1`.

If unsuccessful for another reason, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`	An internal error code that indicates communications have not been initialized.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncUnRegisterConduit` function to unregister a conduit that is currently registered. This function determines if the handle is to a registered conduit, and if so, unregisters the conduit. If `handle` is not a handle to a registered conduit, `SyncUnRegisterConduit` returns an error code.

See Also The SyncRegisterConduit function.

SyncWriteAppPreference

Purpose Writes application preference information into a preferences database on the handheld.

Compatibility

Palm OS version	Sync Manager version
2.0 or later	All

Prototype

```
long SyncWriteAppPreference(CRawPreferenceInfo&
rInfo);
```

Parameters

--> `rInfo` An object of The CRawPreferenceInfo Class, which contains information about the application preference. Use the following fields with this function:

--> `m_pBytes`
The data buffer in which you have stored the preference information.

--> `m_version`
The version number, as assigned by the application.

--> `m_creator`
The 4-byte creator ID of the preference block; this is usually the same as the application's creator ID.

--> `m_prefId`
The 2-byte ID of the preference block that you want to write.

--> `m_backedUp`
A Boolean value; if this is `TRUE`, the block is written to the Saved preferences database; if this is `FALSE`, the block is written to the Unsaved preferences database.

--> `m_nBytes`
The size of the preference block stored in the `m_pBytes` buffer.

--> `m_dwReserved`
Reserved for future use. You must set this to `NULL` (`0`) before calling this function.

--> `m_reqBytes`
Ignored.

--> `m_retBytes`
Ignored.

--> `m_actSize`
Ignored.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_UNKNOWN_REQUEST

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncWriteAppPreference` function to update an application's preferences block on the handheld.

To use this function, allocate the `m_pBytes` buffer in an object of The CRawPreferenceInfo Class, store the preference information into that buffer, and fill in the following other fields in the object: `m_version`, `m_creator`, `m_prefId`, `m_backedUp`, and `m_nBytes`.

The structure of the data in the preferences block is application dependent. The Sync Manager does not modify this data in any way when sending it. This means that multi-byte integer data is stored using big-endian byte ordering, with the most significant byte stored at the lower address in memory. Your conduit is responsible for performing any necessary byte swapping.

WARNING! You must be sure that the version and size of the preference block is compatible with the version of the application running on the handheld. Writing an improperly sized preference block can corrupt a database.

See Also The SyncReadAppPreference function.

SyncWriteDBAppInfoBlock

Purpose Writes an application info block to an open record or resource database on the handheld. The database must be opened for reading and writing.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncWriteDBAppInfoBlock(BYTE fHandle, CDbGenInfo &rInfo)`

Parameters

--> `fHandle`	A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions.
--> `rInfo`	An object of The CDbGenInfo Class, which contains the information block. Use the following fields with this function:

--> `m_pBytes`
The data buffer in which you have stored the application information.

--> `m_TotalBytes`
The size of the application information block stored in the the `m_pBytes` array.

--> `m_BytesRead`
The size of the application information block stored in the the `m_pBytes` array.

--> `m_dwReserved`
Set to `0`.

--> `m_FileName`
Ignored.

Result If successful, returns `0`, which means that the block was written to the handheld database.

If unsuccessful, returns one of the following non-zero error code values:

```
SYNCERR_COMM_NOT_INIT
SYNCERR_LOST_CONNECTION
SYNCERR_REMOTE_SYS
SYNCERR_REMOTE_MEM
SYNCERR_REMOTE_BAD_ARG
SYNCERR_READ_ONLY
SYNCERR_NO_FILES_OPEN
```

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncWriteDBAppInfoBlock` function to write an application information block to a database on the handheld. For more information about application information blocks, see The File Header Information Blocks.

To use this function, allocate the `m_pBytes` buffer in an object of The CDbGenInfo Class and store your information into that buffer. You then set the `m_TotalBytes` and `m_BytesRead` fields to the size of your data buffer.

NOTE: Due to a problem in earlier versions of the Sync Manager API, you must assign the size of the information block to both the `m_BytesRead` and `m_TotalBytes` fields of your `CDbGenInfo` object before calling the `SyncWriteDbAppInfoBlock` function.

To completely delete the application information block from the handheld database, set both `m_TotalBytes` and `m_BytesRead` to `0`.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadDBAppInfoBlock function.

SyncWriteDBSortInfoBlock

Purpose Writes a sort information block to an open record or resource database on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype

```
long SyncWriteDBSortInfoBlock(BYTE fHandle,
CDbGenInfo &rInfo)
```

Parameters

--> `fHandle` A handle to the database on the handheld. This handle is returned by a call to the `SyncOpenDB` or `SyncCreateDB` functions. The database must be opened for reading and writing.

--> `rInfo` An object of The CDbGenInfo Class, which contains the record ID sorting information. Use the following fields with this function:

--> `m_pBytes`
The data buffer in which you have stored the sorted record IDs.

--> `m_TotalBytes`
The number of bytes in the `m_pBytes` data array.

--> `m_dwReserved`
Set to `0`.

Result If successful, returns `0`, which means the block was written to the handheld database.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`

`SYNCERR_LOST_CONNECTION`

`SYNCERR_REMOTE_SYS`

`SYNCERR_REMOTE_MEM`

`SYNCERR_REMOTE_BAD_ARG`

`SYNCERR_READ_ONLY`

`SYNCERR_NO_FILES_OPEN`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncWriteDBSortInfoBlock` function to write a sorting information block to a database on the handheld. For more information about sorting blocks, see The File Header Information Blocks.

To use this function, allocate the `m_pBytes` buffer in an object of The CDbGenInfo Class and store the sorting information in that buffer. You then must assign the size of your buffer to the `m_TotalBytes` field.

To completely delete the sorting information block from the handheld database, set `m_TotalBytes` to `0`.

For performance optimization information, see The Sync Manager and Performance.

See Also The SyncReadDBSortInfoBlock function.

SyncWriteRec

Purpose Writes a record to an open record database on the handheld.

Compatibility

Palm OS version	Sync Manager version	Notes
All	All	See Result section for version-based behavior differences.

Prototype `long SyncWriteRec(CRawRecordInfo &rInfo)`

Parameters

<-> `rInfo` An object of The CRawRecordInfo Class, which contains information about the record and database. Use the following fields with this function:

--> `m_pBytes`
The record to be written.

--> `m_FileHandle`
The handle to an open record database, which must be open for reading and writing.

<-> `m_RecId`
The record ID. To update or restore an existing record, specify the record's ID.

To add a new record, specify `0`. Upon return, `m_RecId` will contain the new ID for the record.

--> `m_Attribs`
Record attributes. This is a combination of the values described in Record Attributes (eSyncRecAttrs) Constants.

--> `m_CatId`
The record's category index. By convention, use `0` to indicate the unfiled category, or use a value between `1` and `15` for other filed categories.

--> `m_RecSize`
The number of bytes in the record.

--- `m_TotalBytes`
Ignored.

--> `m_dwReserved`
Set to `0`.

--> `m_recIndex`
Ignored.

--> `m_conduitId`
Ignored.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`
`SYNCERR_BAD_OPERATION`
`SYNCERR_READ_ONLY`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncWriteRec` function to write a record to a database on the handheld.

- You can overwrite an existing record by specifying its record ID.
- You can add a new record by supplying `0` as the record ID. Note that the application on the handheld is responsible for assigning new record IDs.

To use this function, allocate the `m_pBytes` buffer in object of The CRawRecordInfo Class and store the record information in that buffer. You then must assign the size of your buffer to the `m_RecSize` field, and fill in the remaining fields with information about the record.

You cannot specify the position in the database of a new record, nor can you rely on a new record being added at the end of the database. Upon completion of synchronization operations, applications on the handheld are sent a `sysAppLaunchCmdSyncNotify` notification, which allows them to sort or update the database as required.

When you write a record to a handheld that is running a version of the Palm OS earlier than version 3.0, the record is always marked as modified. For Palm OS version 3.0, the record is only marked as modified if the `eRecAttrDirty` flag is set in the `m_Attribs` field of `r_Info`.

If you are synchronizing with a built-in (ROM) database on a handheld running a version of the Palm OS earlier than version 2.0, you need to handle a special condition that occurs after a hard reset is performed on the handheld. On these handhelds, the unique record ID seeds generated for record databases are always the same. To avoid record ID collisions, the default conduits zero out existing record IDs on the desktop before restoring the databases on hard-reset handhelds, which forces new, unique record IDs to be generated. You only need to apply this work-around if you are synchronizing with a ROM-based application on a handheld running version 2.0 or earlier of the Palm OS.

When you use a version of the Sync Manager API earlier than version 2.0 to write a record to a database on the handheld, the current iteration index is not updated. For more information, see Modifying a Database While Iterating.

See Also The SyncReadRecordById and SyncReadRecordByIndex functions.

SyncWriteResourceRec

Purpose Writes a resource to an open resource database on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncWriteResourceRec(CRawRecordInfo &rRec)`

Parameters

--> `rRec` An object of The CRawRecordInfo Class, which contains information about the record and database. Use the following fields with this function:

--> `m_pBytes`
The record to be written.

--> `m_FileHandle`
The handle to the open resource database. The database must be open for reading and writing.

--> `m_RecId`
The 4-byte resource type.

--> `m_RecIndex`
The 2-byte resource ID.

--> `m_RecSize`
The resource data size, in bytes.

--> `m_dwReserved`
Reserved for future use. You must set this to `NULL` (`0`) before calling this function.

--> `m_TotalBytes`
Ignored.

--> `m_Attribs`
Ignored.

--> `m_CatId`
Ignored.

--> `m_ConduitId`
Ignored.

Result If successful, returns `0`, which means that the resource was written.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_MEM`
`SYNCERR_REMOTE_BAD_ARG`
`SYNCERR_NO_FILES_OPEN`
`SYNCERR_BAD_OPERATION`
`SYNCERR_READ_ONLY`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncWriteResourceRec` function to write a resource record to a resource database on the handheld.

To use this function, allocate the `m_pBytes` buffer in an object of The CRawRecordInfo Class, and store the resource information in that buffer. You then must assign the size of the resource to the `m_RecSize` field, and fill in the remaining fields with information about the resource.

See Also The SyncReadResRecordByIndex function.

SyncWriteSysDateTime

Purpose Sets the system date and time on the handheld.

Compatibility

Palm OS version	Sync Manager version
All	All

WARNING! Although this function is available in all versions of the Sync Manager API, it does not work properly in any version earlier than version 2.2.

Prototype

```
long SyncWriteSysDateTime(long  lDate);
```

Parameters

--> `lDate`	A `time_t` value that specifies the system date and time value. This value must be in the format returned by the `time` function.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`
`SYNCERR_LOST_CONNECTION`
`SYNCERR_REMOTE_SYS`
`SYNCERR_REMOTE_BAD_ARG`

If you call this function in a version of the Sync Manager API earlier than version 2.2, the `SYNCERR_REMOTE_BAD_ARG` error code is returned.

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You can use the `SyncWriteSysDateTime` function to set the current date and time on the handheld. In general, conduits should avoid changing the system date and time.

IMPORTANT: This function is implemented using POSIX time functions in CodeWarrior's MSL library. It's important to note that starting with CodeWarrior Pro 6, Metrowerks changed this function to use an epoch of 1970 instead of 1900. HotSync Manager 3.0 is built using CodeWarrior 7.2, so it's possible your conduit may need to be rebuilt with the new version of MSL to avoid being off by 70 years.

The `SyncWriteSysDateTime` function does not notify applications on the handheld that it has changed the time. Some applications, such as the built-in datebook, need to know when the system time changes so that they can adjust their alarm settings. To work around this problem, you need to call the SyncRebootSystem function, which will cause a soft-reset of the handheld after HotSync completes. All applications on the handheld are notified of the reset and can make any necessary adjustments.

See Also The SyncReadSysDateTime function.

SyncYieldCycles

Purpose Processes events for the HotSync application, which allows the HotSync progress indicator to be updated for the user.

Compatibility

Palm OS version	Sync Manager version
All	All

Prototype `long SyncYieldCycles(WORD wMaxMiliSecs);`

Parameters

--> `wMaxMiliSecs`	The maximum number of milliseconds to spend servicing events. This value is currently ignored; you should supply a value of `1`.

Result If successful, returns `0`.

If unsuccessful, returns one of the following non-zero error code values:

`SYNCERR_COMM_NOT_INIT`

For more information about the error codes, see Sync Manager Error Code Summary.

Comments You call the `SyncYieldCycles` function periodically to maintain a connection with the handheld, to keep the HotSync progress display indicator current, and to make it possible for the user to press HotSync Manager's Cancel button. If you neglect to call this function frequently enough, the user interface will appear to be frozen.

NOTE: You should call this function as frequently as possible. You can do so without appreciable performance penalty.

You must call this function at least once every seven seconds during times of communication inactivity. Otherwise, the handheld might terminate the connection prematurely.

When you call this function, the Sync Manager drains the message queue and pings the handheld. Since there are usually no messages to be processed, `SyncYieldCycles` usually returns immediately.

Your calls to the `SyncYieldCycles` function must be made from the same thread that started your conduit.

Sync Manager Error Code Summary

Table 2.1 describes the error codes that you can receive from the Sync Manager functions. These error codes are declared in the `syncmgr.h` header file.

Table 2.1 Sync Manager Errors

Error Code	Description
`-1`	A non-specific error occurred.
`SYNCERR_ARG_MISSING`	An internal Desktop Link error that indicates a protocol implementation error.
`SYNCERR_BAD_ARG`	An invalid parameter has been passed to a function, or the parameter is too large.
`SYNCERR_BAD_ARG_WRAPPER`	An internal Desktop Link error that indicates a protocol implementation error
`SYNCERR_BAD_OPERATION`	The requested operation is not supported on the given database type (record or resource).
`SYNCERR_COMM_NOT_INIT`	An internal error code that indicates communications have not been initialized.

Table 2.1 Sync Manager Errors *(continued)*

Error Code	Description
`SYNCERR_FILE_ALREADY_EXIST`	The database could not be created because another database with the same name already exists on the handheld.
`SYNCERR_FILE_ALREADY_OPEN`	The requested database is already opened.
`SYNCERR_FILE_NOT_OPEN`	The attempt to open the database failed.
`SYNCERR_FILE_OPEN`	Not used.
`SYNCERR_LIMIT_EXCEEDED`	A data limit has been exceeded on the handheld. For example, this happens when the HotSync error log size limit has been exceeded on the handheld.
`SYNCERR_LOCAL_BUFF_TOO_SMALL`	The passed buffer is too small for the reply data.
`SYNCERR_LOCAL_CANCEL_SYNC`	HotSync was cancelled by the desktop computer user.
`SYNCERR_LOCAL_MEM`	A memory allocation error occurred on the desktop computer.
`SYNCERR_LOST_CONNECTION`	The connection with the handheld was lost.
`SYNCERR_MORE`	Not used.
`SYNCERR_NO_FILES_OPEN`	An operation was requested on a database, and there are not any open databases.
`SYNCERR_NONE`	The function call succeeded, without error.

Table 2.1 Sync Manager Errors *(continued)*

Error Code	Description
`SYNCERR_NOT_FOUND`	The requested database, record, resource, etc. could not be found. This error code replaces the earlier `SYNCERR_FILE_NOT_FOUND` error code. NOTE: This result code is returned when iterating through a database to indicate that there are no more records to retrieve.
`SYNCERR_READ_ONLY`	Your function does not have write access to the database, or the database is in ROM. This error code replaces the earlier `SYNCERR_ROM_BASED` error code.
`SYNCERR_RECORD_BUSY`	The requested record is in use by someone else and will remain so indefinitely.
`SYNCERR_RECORD_DELETED`	The requested record has either been deleted or archived.
`SYNCERR_REMOTE_BAD_ARG`	An invalid argument has been passed to the handheld.
`SYNCERR_REMOTE_CANCEL_SYNC`	HotSync was cancelled by the handheld user.
`SYNCERR_REMOTE_MEM`	There is insufficient memory on the handheld to receive or complete the request.
`SYNCERR_REMOTE_NO_SPACE`	There is insufficient memory in the data store on the handheld to complete the request. This generally occurs when attempting to write a record or resource to a handheld database.
`SYNCERR_REMOTE_SYS`	A generic system error on the handheld.This is returned when the exact cause is unknown.

Table 2.1 Sync Manager Errors *(continued)*

Error Code	Description
`SYNCERR_TOO_MANY_OPEN_FILES`	Request failed because there are too many open databases (for efficiency, the current Desktop Link implementation supports only one open database at a time). This error code replaces the earlier `SYNCERR_TOO_MANY_FILES` error code.
`SYNCERR_UNKNOWN`	An unknown error occurred: the handheld error code could not be mapped into a desktop error code.
`SYNCERR_UNKNOWN_REQUEST`	This request (command) is not supported by the handheld.

3

HotSync Log API

This chapter describes the HotSync® log API, which you can use to access the HotSync log.

HotSync Log Constants

This section describes the constants that you can use with the HotSync log functions.

Log Activity Type Constants

The log activity type constants specify the action that caused the entry to be added to the log.

```
enum Activity {
              slText = -1,
              slDoubleModify,
              slDoubleModifyArchive,
              slReverseDelete,
              slTooManyCategories,
              slCategoryDeleted,
              slDateChanged,
              slCustomLabel,
              slChangeCatFailed,
              slRemoteReadFailed,
              slRemoteAddFailed,
              slRemotePurgeFailed,
              slRemoteChangeFailed,
              slRemoteDeleteFailed,
              slLocalAddFailed,
              slRecCountMismatch,
              slXMapFailed,
              slArchiveFailed,
              slLocalSaveFailed,
              slResetFlagsFailed,
                     slSyncStarted,
                     slSyncFinished,
                     slSyncAborted,
                     slWarning,
                     slDoubleModifySubsc,
                     slSyncDidNothing
   };
```

`slText`	Logging a text entry.
`slDoubleModify`	A record has been modified on both the desktop computer and handheld.
`slDoubleModifyArchive`	A record that has been modified on both the desktop and the handheld and has been archived.
`slReverseDelete`	
`slTooManyCategories`	No more categories can be added.
`slCategoryDeleted`	A category was deleted.
`slDateChanged`	The date was changed.
`slCustomLabel`	
`slChangeCatFailed`	Changing a category failed.
`slRemoteReadFailed`	Reading a record failed on the handheld.
`slRemoteAddFailed`	Adding a record on the handheld failed.
`slRemotePurgeFailed`	Purging a record on the handheld failed.
`slRemoteChangeFailed`	Changing a record on the handheld failed.

`slRemoteDeleteFailed`	Deleting a record on the handheld failed.
`slLocalAddFailed`	Adding a record on the desktop computer failed.
`slRecCountMismatch`	Record counts did not match.
`slXMapFailed`	The position cross-map operation failed.
`slArchiveFailed`	The archive operation failed.
`slLocalSaveFailed`	Saving data on the desktop computer failed.
`slResetFlagsFailed`	Resetting of the synchronization flags failed.
`slSyncStarted`	The synchronization operation started.
`slSyncFinished`	The synchronization operation finished successfully.
`slSyncAborted`	The synchronization operation was aborted.
`slWarning`	Logging a warning.
`slDoubleModifySubsc`	
`slSyncDidNothing`	The synchronization operation did not perform any actions.

HotSync Log Function Summary

This section describes the `LogAddEntry` function, which is the only function implemented in the Macintosh CDK for use with the HotSync log.

LogAddEntry

Purpose Adds an entry to the log.

Prototype

```
long LogAddEntry(LPCTSTR pszEntry, Activity act,
BOOL bTimeStamp);
```

Parameters

--> `pszEntry`	The string to enter into the log.
--> `act`	The activity type for the log entry. Use one of the constants described in Log Activity Type Constants.
--> `bTimeStamp`	A Boolean value. If this is true, the log entry is time-stamped.

Result If successful, returns `0`.

If unsuccessful, returns a standard Macintosh toolbox error code.

Comments You can call the `LogAddEntry` function to add an entry to the HotSync log. The entry can optionally be time-stamped in the log.

NOTE: The remainder of the functions in this chapter are included in the CDK headers, but are not implemented for use on the Macintosh. These functions are intended for use by developers who are creating applications similar to the HotSync Manager application.

Unimplemented HotSync Log Functions

This section describes the log functions that are not implemented on the Macintosh. These functions are used to create applications like the HotSync Manager application, and are provided for Windows developers. Conduit developers never use these functions.

- The `LogBuildRemoteLog` method builds the synchronization log for the handheld.
- The `LogCloseLog` method closes the HotSync log.

- The `LogGetWorkFileName` method returns the name of the file that the HotSync Manager application is using as the working log file name.
- The `LogInit` method initializes the log.
- The `LogSaveLog` method saves the HotSync log to a file.
- The `LogTestCounters` method determines if any errors were logged.
- The `LogUnInit` method destroy the log object.

For more information on these logging functions, see the Windows Conduit Development Kit documentation.

4

Expansion Manager API

The Expansion Manager on the handheld is an optional system extension that adds support for hardware expansion cards on Palm Powered™ handhelds. The handheld Expansion Manager's primary function is to manage slots on the handheld and the drivers associated with those slots. Individual slot drivers on the handheld — which are provided by handheld manufacturers — provide support for various expansion card types including Secure Digital (SD), MultiMediaCard (MMC), CompactFlash, Sony's Memory Stick, and others.

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

This chapter provides the following information about the Expansion Manager API:

- Expansion Manager Constants
- Expansion Manager Data Structures
- Expansion Manager Functions
- Expansion Manager Error Codes

Chapter 8, "Using Expansion Technology," on page 77The desktop Expansion Manager functions are available in `HotSync Libraries` and declared in `ExpansionMgr.h`. (Expansion Manager error codes are declared in `VFSErr.h`.) For more information on the Expansion Manager, see Chapter 8, "Using Expansion Technology," on page 77 of the *C/C++ Sync Suite Companion for Macintosh*.

NOTE: The Expansion Manager is an optional system extension on handhelds. Therefore you should check for the presence of the Expansion Manager on the handheld before calling any Expansion Manager API functions. See "Verifying Handheld Compatibility" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*.

Expansion Manager Constants

The following types of constants are defined for the Expansion Manager:

- Hardware Capability Flags
- Maximum Length of ExpCardInfoType String
- Defined Media Type Constants
- Directory Entry Iterator Start/Stop Constants

Hardware Capability Flags

The following flags describe the capabilities of the card in `ExpCardInfoType.capabilityFlags`.

`expCapabilityHasStorage`
The card has data storage. The `expCapabilityReadOnly` flag specifies whether the card can be written or only read, though.

`expCapabilityReadOnly`
The card is read-only.

Maximum Length of ExpCardInfoType String

The `expCardInfoStringMaxLen` constant defines the maximum length of a string in a member of the `ExpCardInfoType` structure.

Defined Media Type Constants

Table 4.1 defines the constants for the media types supported by the Expansion Manager. These media types are used with the `ExpSlotMediaType` function and the `VFSVolumeInfo` function in the `VolumeInfoType.mediaType` field.

Table 4.1 Media types defined by the Expansion Manager

Constant	Value	Description
`ExpMediaType_Any`	`'wild'`	Matches all media types when looking up a default directory
`ExpMediaType_MemoryStick`	`'mstk'`	Memory Stick
`ExpMediaType_CompactFlash`	`'cfsh'`	CompactFlash
`ExpMediaType_SecureDigital`	`'sdig'`	Secure Digital
`ExpMediaType_MultiMediaCard`	`'mmcd'`	MultiMediaCard
`ExpMediaType_SmartMedia`	`'smed'`	SmartMedia
`ExpMediaType_RAMDisk`	`'ramd'`	A RAM disk based media
`ExpMediaType_PoserHost`	`'pose'`	Host file system emulated by Palm OS® Emulator
`ExpMediaType_PlugNPlay`	`'pnps'`	Universal “plug and play” (PnP) connector

Directory Entry Iterator Start/Stop Constants

Table 4.2 defines the constants that control when to start and stop the iterated calls to the `VFSDirEntryEnumerate` function.

Table 4.2 Directory entry iterator start/stop constants

Constant	Value	Description
`expIteratorStart`	`0L`	Before the first call, initialize the iterator to this value.
`expIteratorStop`	`0xffffffffL`	The function returns this value when it returns information about the last entry in the directory.

Expansion Manager Data Structures

This section describes the `ExpCardInfoType` structure defined for the Expansion Manager.

ExpCardInfoType

The `ExpCardInfoType` declaration defines a structure that is passed to `ExpCardInfo`. This structure is used to determine the characteristics of the card loaded in the slot. It is initialized by the underlying slot driver with the following information.

```
typedef struct ExpCardInfoTag {
 UInt32 capabilityFlags;
 Char manufacturerStr[expCardInfoStringMaxLen+1];
 Char productStr[expCardInfoStringMaxLen+1];
 Char deviceClassStr[expCardInfoStringMaxLen+1];
 Char deviceUniqueIDStr[expCardInfoStringMaxLen+1];
} ExpCardInfoType, *ExpCardInfoPtr;
```

Field Descriptions

`capabilityFlags`	Describes the capabilities of the card. This is set to one or more of the Hardware Capability Flags.
`manufacturerStr`	Names the manufacturer of the card — for example, “Palm” or “Motorola”.

`productStr`	Name of the product. For example “SafeBackup 32 MB”.
`deviceClassStr`	Describes the type of card — for example, “Backup” or “Ethernet”.
`deviceUniqueIDStr`	Unique identifier for the product — for example, a serial number. This value is set to the empty string if no identifier exists.

Expansion Manager Functions

The following Expansion Manager functions are defined in this section:

- `ExpCardInfo`
- `ExpCardPresent`
- `ExpSlotEnumerate`
- `ExpSlotMediaType`

ExpCardInfo

Purpose Retrieves information about an expansion card in a given slot.

Declared In `ExpansionMgr.h`

Prototype

```
long ExpCardInfo (WORD slotRefNumber,
ExpCardInfoType *pCardInfo, void *pVoid)
```

Parameters

`-> slotRefNumber`	The slot reference number passed back by `ExpSlotEnumerate`.
`<- pCardInfo`	Pointer to `ExpCardInfoType` structure that contains information about the card in the specified slot.
`<-> pVoid`	This parameter is unused in this version of the Expansion Manager. Pass in `NULL` and ignore the value passed back.

Result If successful, returns `SYNCERR_NONE`.

If unsuccessful, returns one of the following error codes:

```
expErrCardNoSectorReadWrite
expErrCardNotPresent
expErrInvalidSlotRefNumber
expErrSlotDeallocated
expErrUnsupportedOperation
```

For more information about the error codes, see "Expansion Manager Error Codes" on page 171.

Comments This routine returns information about a card, including whether the card supports secondary storage or is strictly read-only, by filling in the `ExpCardInfoType` structure's fields.

Compatibility Palm OS version: 4.0.
Expansion Manager version: All.
See "Checking for Expansion Cards" on page 87 in the C/C++ *Sync*

Suite Companion for Macintosh for ways to confirm the presence of the Expansion Manager on the handheld.

See Also `ExpCardPresent`, `ExpSlotEnumerate`

ExpCardPresent

Purpose Determines whether a card is present in the given slot.

Declared In `ExpansionMgr.h`

Prototype

```
long ExpCardPresent (WORD slotRefNumber,
void *pVoid)
```

Parameters

`-> slotRefNumber`	The slot reference number passed back by `ExpSlotEnumerate`.
`<-> pVoid`	This parameter is unused in this version of the Expansion Manager. Pass in `NULL` and ignore the value passed back.

Result If successful, returns `SYNCERR_NONE`.

If unsuccessful, returns one of the following error codes:

```
expErrCardNotPresent
expErrInvalidSlotRefNumber
expErrSlotDeallocated
expErrUnsupportedOperation
```

For more information about the error codes, see “Expansion Manager Error Codes” on page 171.

Comments Call this function to test whether a card is present in a slot before making any VFS Manager API calls to access files on a card.

Compatibility Palm OS version: 4.0.
Expansion Manager version: All.

See "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh* for ways to confirm the presence of the Expansion Manager on the handheld.

See Also `ExpCardInfo`, `ExpSlotEnumerate`

ExpSlotEnumerate

Purpose Enumerates the valid slots to obtain a list of slot reference numbers.

Declared In `ExpansionMgr.h`

Prototype

```
long ExpSlotEnumerate
(WORD *pNumSlotRefListEntires,
WORD *pSlotRefNumList, void *pVoid)
```

Parameters

`<-> pNumSlotRefListEntires`
On entry, a pointer to the number of slot reference numbers allocated. On return, it is a pointer to the number of slot reference numbers filled into `pSlotRefNumList`.

`<- pSlotRefNumList`
A pointer to an array of slot reference numbers. The caller must allocate this buffer before calling this function.

`<-> pVoid`
This parameter is unused in this version of the Expansion Manager. Pass in `NULL` and ignore the value passed back.

Result If successful, returns `SYNCERR_NONE`.

If unsuccessful, returns one of the following error codes:

`expErrUnsupportedOperation`

For more information about the error codes, see "Expansion Manager Error Codes" on page 171.

Comments This function passes back a list of slot reference numbers for slots on the handheld. Note that you must allocate sufficient space for `pSlotRefNumList` before calling this function.

Example The following example shows a way to allocate sufficient space before calling `ExpSlotEnumerate`.

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

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

Compatibility Palm OS version: 4.0.
Expansion Manager version: All.

See "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh* for ways to confirm the presence of the Expansion Manager on the handheld.

See Also `ExpCardInfo`, `ExpCardPresent`

ExpSlotMediaType

Purpose Obtains the media type identifier for the specified slot.

Declared In `ExpansionMgr.h`

Prototype
```
long ExpSlotMediaType (WORD slotRefNum,
UINT32 *pui32SlotMediaType)
```

Parameters

-> `slotRefNum` The slot reference number (passed back by `ExpSlotEnumerate`.) for which to determine the media type.

<- `pui32SlotMediaType` A pointer to a `UINT32` that identifies the media type of the specified slot. The section "Defined Media Type Constants" on page 163 lists the possible values.

Result If successful, returns `SYNCERR_NONE`.

If unsuccessful, returns one of the following error codes:

```
expErrCardNotPresent
expErrSlotDeallocated
expErrUnsupportedOperation
```

For more information about the error codes, see "Expansion Manager Error Codes" on page 171.

Compatibility Palm OS version: 4.0.
Expansion Manager version: All.

See "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh* for ways to confirm the presence of the Expansion Manager on the handheld.

See Also `ExpCardInfo`, `ExpSlotEnumerate`

Expansion Manager Error Codes

Table 4.3 describes the error codes that the Expansion Manager functions can return. These error codes are declared in the `VFSErr.h` header file.

Table 4.3 Expansion Manager error codes

Constant	Description
`expErrCardNoSectorReadWrite`	The card does not support the slot driver block read/write API.
`expErrCardNotPresent`	No card is present in the given slot.
`expErrEnumerationEmpty`	No volumes are present to enumerate or none remain to enumerate.
`expErrInvalidSlotRefNumber`	The slot reference number is not valid.
`expErrNotEnoughPower`	Insufficient battery power on the handheld to perform the operation.
`expErrNotOpen`	The file system library on the handheld necessary for this call has not been installed or has not been opened.
`expErrSlotDeallocated`	The slot reference number is within the valid range, but the Expansion Manager has unloaded the slot driver on the handheld.
`expErrUnsupportedOperation`	The operation is unsupported or undefined.

5

Virtual File System Manager API

The Virtual File System (VFS) Manager is a layer of software that allows conduits to access all installed file systems on handheld expansion cards. It provides a unified API to conduit developers while allowing them to seamlessly access many different types of file systems — such as VFAT, HFS, and NFS — on many different types of media, including Secure Digital (SD), MultiMediaCard (MMC), CompactFlash, Sony's Memory Stick, and others.

This chapter provides reference material for the VFS Manager API as follows:

- VFS Manager Constants
- VFS Manager Data Structures
- VFS Manager Functions
- VFS Manager Error Codes

The VFS Manager functions are available in `HotSync Libraries` and declared in `VFSMgr.h`. (VFS Manager error codes are declared in `VFSErr.h`.) For more information on the VFS Manager, see Chapter 8, "Using Expansion Technology," on page 77 in the *C/C++ Sync Suite Companion*.

NOTE: The VFS Manager is an optional system extension on handhelds. Therefore you should check for the presence of the VFS Manager on the handheld before call any VFS Manager API functions. See "Verifying Handheld Compatibility" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*.

VFS Manager Constants

The following types of constants are defined for the VFS Manager:

- Supported File Systems Constants
- Open Mode Constants
- File and Directory Attributes
- Volume Attributes
- Volume Mount Class Constants
- Invalid Reference Constants
- File Origin Constants
- Date Type Constants

Supported File Systems Constants

The file systems in Table 5.1 are currently supported by the VFS Manager. These values are used with `VFSVolumeInfo` in the `VolumeInfoType.fsType` parameter.

Table 5.1 Supported file systems constants

Constant	Value	Description
`fsFilesystemType_VFAT`	`'vfat'`	FAT12 and FAT16, extended to handle long filenames.
`fsFilesystemType_FAT`	`'fats'`	FAT12 and FAT16, which handles only 8.3 filenames.
`fsFilesystemType_NTFS`	`'ntfs'`	Windows NT file system.
`fsFilesystemType_HFSPlus`	`'hfse'`	Macintosh extended hierarchical file system.
`fsFilesystemType_HFS`	`'hfss'`	Macintosh standard hierarchical file system.

Table 5.1 Supported file systems constants *(continued)*

Constant	Value	Description
`fsFilesystemType_MFS`	`'mfso'`	Macintosh original file system.
`fsFilesystemType_EXT2`	`'ext2'`	Linux file system.
`fsFilesystemType_FFS`	`'ffsb'`	Unix Berkeley block based file system.
`fsFilesystemType_NFS`	`'nfsu'`	Unix Networked file system.
`fsFilesystemType_AFS`	`'afsu'`	Unix Andrew file system
`fsFilesystemType_Novell`	`'novl'`	Novell file system.
`fsFilesystemType_HPFS`	`'hpfs'`	OS/2 High Performance file system

Open Mode Constants

Table 5.2 describes constants that are used for the `openMode` parameter to the `VFSFileOpen` function. These constants specify the mode in which a file or directory is opened.

Table 5.2 Open mode constants

Constant	Value	Description
`vfsModeExclusive`	`(0x0001UL)`	Open and lock the file or directory. This mode excludes anyone else from using the file or directory until it is closed.
`vfsModeRead`	`(0x0002UL)`	Open for read access.
`vfsModeWrite`	`(0x0004UL | vfsModeExclusive)`	Open for exclusive write access. This mode excludes anyone else from using the file or directory until it is closed.

Table 5.2 Open mode constants *(continued)*

Constant	Value	Description
`vfsModeCreate`	`(0x0008U)`	Create the file if it doesn't already exist.
`vfsModeTruncate`	`(0x0010U)`	Truncate the file to zero bytes after opening, removing all existing data.
`vfsModeReadWrite`	`(vfsModeWrite | vfsModeRead)`	Open for read/write access.

File and Directory Attributes

The constants in Table 5.3 define bits that can be used individually or in combination when setting or interpreting the file attributes for a given file or directory. See `VFSFileGetAttributes`, `VFSFileSetAttributes`, and the `FileInfoType` data structure for specific use.

Table 5.3 File and directory attributes

Constant	Value	Description
`vfsFileAttrReadOnly`	`(0x00000001UL)`	Read-only file or directory
`vfsFileAttrHidden`	`(0x00000002UL)`	Hidden file or directory
`vfsFileAttrSystem`	`(0x00000004UL)`	System file or directory
`vfsFileAttrVolumeLabel`	`(0x00000008UL)`	Volume label
`vfsFileAttrDirectory`	`(0x00000010UL)`	Directory
`vfsFileAttrArchive`	`(0x00000020UL)`	Archived file or directory
`vfsFileAttrLink`	`(0x00000040UL)`	Link to another file or directory

Volume Attributes

The constants in Table 5.4 define bits that can be used individually or in combination to make up the attributes field in the `VolumeInfoType` structure.

Table 5.4 Volume attributes

Constant	Value	Description
`vfsVolumeAttrSlotBased`	`(0x00000001UL)`	The volume is associated with a slot driver as opposed to the 68K Palm™ Simulator or Palm OS® Emulator.
`vfsVolumeAttrReadOnly`	`(0x00000002UL)`	The volume is read only.
`vfsVolumeAttrHidden`	`(0x00000004UL)`	The volume should not be visible to the user. For more information, see "Hidden Volumes" on page 95 in the *C/C++ Sync Suite Companion for Macintosh*.

Volume Mount Class Constants

The constants in Table 5.5 define how a given volume is mounted. The `mountClass` field in the `VFSAnyMountParamType` and `VolumeInfoType` structures takes one of these values.

Table 5.5 Volume mount class constants

Constant	Value	Description
`vfsMountClass_SlotDriver`	`sysFileTSlotDriver`	Mount the volume with a slot driver shared library.
`vfsMountClass_Simulator`	`sysFileTSimulator`	Mount the volume through the 68K Palm Simulator. This is used for testing.
`vfsMountClass_POSE`	`'pose'`	Mount the volume through Palm OS® Emulator. This is used for testing.

Invalid Reference Constants

The constants in Table 5.6 are placeholders for when you do not have a valid reference number.

Table 5.6 Invalid reference constants

Constant	Value	Description
`vfsInvalidVolRef`	`0`	The volume has not been formatted. See `VFSVolumeFormat`.
`vfsInvalidFileRef`	`0`	The file reference number is invalid.
`vfsInvalidSlotLibRefNum`	`-1`	The slot library reference number is not available. See `VFSSlotMountParamType`.

File Origin Constants

The constants in Table 5.7 define the origins of relative offsets passed to the `VFSFileSeek` function.

Table 5.7 File origin constants

Constant	Value	Description
`fsOriginBeginning`	0	From the beginning (first data byte of file).
`fsOriginCurrent`	1	From the current position.
`fsOriginEnd`	2	From the end of the file (one position beyond last data byte). Only negative offsets are legal from this origin.

Date Type Constants

The constants in Table 5.8 define the types of dates you can specify with the `VFSFileGetDate` and `VFSFileSetDate` functions.

Table 5.8 Date type constants

Constant	Value	Description
`vfsFileDateCreated`	1	The date the file was created.
`vfsFileDateModified`	2	The date the file was last modified.
`vfsFileDateAccessed`	3	The date the file was last accessed.

VFS Manager Data Structures

The following data structures are defined for the VFS Manager:

- `FileInfoType`
- `FileRef`
- `FileOrigin`
- `VFSAnyMountParamType`

- `VFSSlotMountParamType`
- `VolumeInfoType`

FileInfoType

Usage The `FileInfoType` structure contains information about a specified file or directory. This information is passed back as a parameter to `VFSDirEntryEnumerate`. The structure is defined as follows:

Declaration

```
typedef struct FileInfoTag {
  UINT32 attributes;
  char *nameP;
  WORD nameBufLen;
} FileInfoType, *FileInfoPtr;
```

Fields

`attributes`	Characteristics of the file or directory. See "File and Directory Attributes" on page 176 for the bits that make up this field.
`*nameP`	Pointer to the buffer that receives the full name of the file or directory. Allocate a sufficiently large buffer and specify its size in `nameBufLen`.
`nameBufLen`	Size of the `nameP` buffer, in bytes.

FileRef

Usage The `FileRef` type is used to encode references to files and directories.

Declaration

```
typedef UINT32 FileRef;
```

FileOrigin

Usage The `FileOrigin` type is used to calculate the new position from which to read or write with the `VFSFileSeek` function. See "File Origin Constants" on page 179 for descriptions of the supported file origins.

Declaration

```
typedef WORD FileOrigin;
```

VFSAnyMountParamType

Usage The `VFSAnyMountParamType` structure is a base structure for volume mount parameters for different file systems. For slot-based file systems, use `VFSSlotMountParamType`.

Declaration

```
typedef struct VFSAnyMountParamTag {
  WORD volRefNum;
  WORD reserved;
  UINT32 mountClass;
} VFSAnyMountParamType;
typedef VFSAnyMountParamType
*VFSAnyMountParamPtr;
```

Fields

`volRefNum`	The volume reference number. This is initially obtained when you call `VFSVolumeEnumerate` to successfully enumerate volumes.
`reserved`	Reserved for future use.
`mountClass`	Defines the type of mount to use with the specified volume. See "Volume Mount Class Constants" on page 178 for a list of mount types.

VFSSlotMountParamType

Usage The `VFSSlotMountParamType` structure is used when you are mounting a card located in a physical slot. The `vfsMountParam.mountClass` field must be set to `VFSMountClass_SlotDriver`.

Declaration

```
typedef struct VFSSlotMountParamTag {
  VFSAnyMountParamType vfsMountParam;
  WORD slotLibRefNum;
  WORD slotRefNum;
} VFSSlotMountParamType;
```

Fields

`vfsMountParam`	See the description of `VFSAnyMountParamType` for an explanation of the fields in this structure. This is passed back in the `VolumeInfoType` structure in a call to `VFSVolumeInfo`. Set `vfsMountParam.mountClass` to `VFSMountClass_SlotDriver` to mount a physical slot.
`slotLibRefNum`	Reference number for the slot driver library allocated to the given slot number. If this value is not available, set this field to `vfsInvalidSlotLibRefNum`.
`slotRefNum`	The slot reference number obtained by the Expansion Manager's `ExpSlotEnumerate` function.

VolumeInfoType

Usage The `VolumeInfoType` structure defines information that is passed back by `VFSVolumeInfo` and used throughout the VFS Manager functions.

Declaration

```
typedef struct VolumeInfoTag {
  UINT32 attributes;
  UINT32 fsType;
  UINT32 fsCreator;
  UINT32 mountClass;
  WORD   slotLibRefNum;
  WORD   slotRefNum;
  UINT32 mediaType;
  UINT32 reserved;
} VolumeInfoType, *VolumeInfoPtr;
```

Fields

`attributes`	Characteristics of the volume. See “Volume Attributes” on page 177 for the bits that make up this field.
`fsType`	File system type for this volume. See “Supported File Systems Constants” on page 174 for a list of the supported file systems.
`fsCreator`	Creator code of this volume’s file system driver. This information is used with `VFSCustomControl`.
`mountClass`	Mount class of the driver that mounted this volume. The supported mount classes are listed under “Volume Mount Class Constants” on page 178.
`slotLibRefNum`	Reference to the slot driver library with which the volume is mounted. This field is valid only when the `mountClass` is `vfsMountClass_SlotDriver`.

`slotRefNum`	Expansion Manager slot reference number where the card containing the volume is loaded. This field is valid only when the `mountClass` is `vfsMountClass_SlotDriver`.
`mediaType`	Type of card media. See "Defined Media Type Constants" on page 163 for the list of values.
`reserved`	Reserved for future use.

VFS Manager Functions

This section describes all of the following VFS Manager functions in alphabetical order.

`VFSCustomControl`	`VFSFileOpen`	`VFSGetAPIVersion`
`VFSDirCreate`	`VFSFilePut`	`VFSGetDefaultDirectory`
`VFSDirEntryEnumerate`	`VFSFileRead`	`VFSImportDatabaseFromFile`
`VFSExportDatabaseToFile`	`VFSFileRename`	`VFSSupport`
`VFSFileClose`	`VFSFileResize`	`VFSVolumeEnumerate`
`VFSFileCreate`	`VFSFileSetAttributes`	`VFSVolumeFormat`
`VFSFileDelete`	`VFSFileSetDate`	`VFSVolumeGetLabel`
`VFSFileEOF`	`VFSFileSeek`	`VFSVolumeInfo`
`VFSFileGet`	`VFSFileSize`	`VFSVolumeSetLabel`
`VFSFileGetAttributes`	`VFSFileTell`	`VFSVolumeSize`
`VFSFileGetDate`	`VFSFileWrite`	

VFSCustomControl

Purpose Makes a custom API call to a particular file system driver, given the driver's creator ID.

Declared In `VFSMgr.h`

Prototype

```
long VFSCustomControl (UINT32 fsCreator,
UINT32 apiCreator, WORD apiSelector,
void *pDataBuf, WORD *pwDataBufLen)
```

Parameters

`-> fsCreator`	Creator of the file system on the handheld to call. A value of zero tells the VFS Manager to check each registered file system, looking for one that supports the call.
`-> apiCreator`	Registered creator ID of the file system driver on the handheld.
`-> apiSelector`	Code for the custom operation that you want the file system driver to perform. See the file system driver manufacturer for details.
`<-> pDataBuf`	A pointer to a buffer containing data specific to the operation. On exit, depending on the function of the particular custom call and on the value of `valueLenP`, the contents of this buffer may have been updated.
`<-> pwDataBufLen`	On entry, points to the size of the `pDataBuf` buffer. On exit, this value reflects the size of the data written to the `pDataBuf` buffer. If `valueLenP` is `NULL`, `pDataBuf` is passed to the file system but is not updated on exit.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
expErrUnsupportedOperation
SYNCERR_REMOTE_BAD_ARG
```

```
vfsErrInvalidOperation
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments The driver identifies the call and its API by a registered creator ID and a selector. (You can use `VFSVolumeInfo` to determine the creator ID of the file system for a given volume.) This allows file system developers to extend the API by defining selectors for their creator IDs. It also allows file system developers to support selectors (and custom calls) defined by other file system developers.

This function must return `expErrUnsupportedOperation` for all unsupported or undefined opcodes and/or creators.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSVolumeInfo`

VFSDirCreate

Purpose Creates a new directory.

Declared In `VFSMgr.h`

Prototype

```
long VFSDirCreate (WORD volRefNum,
const char *pszDirName)
```

Parameters

`-> volRefNum`	Volume reference number passed back by `VFSVolumeEnumerate`.
`-> pszDirName`	A pointer to the full path of the directory to be created.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrFileAlreadyExists
vfsErrInvalidOperation
vfsErrNoFileSystem
vfsErrVolumeBadRef
vfsErrVolumeFull
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments All parts of the path except the last component must already exist. The `vfsFileAttrDirectory` attribute is set with this function.

`VFSDirCreate` does not open the directory. Any operations you want to perform on this directory require a reference, which is obtained through a call to `VFSFileOpen`.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileOpen`, `VFSFileDelete`, `VFSDirEntryEnumerate`

VFSDirEntryEnumerate

Purpose Enumerates the entries in a given directory. Entries can include files, links, and other directories.

Declared In `VFSMgr.h`

Prototype

```
long VFSDirEntryEnumerate (FileRef dirRef,
UINT32 *pui32DirEntryIterator,
FileInfoType *pFileInfo)
```

Parameters

`-> dirRef` Directory reference passed back by `VFSFileOpen`.

`<-> pui32DirEntryIterator`
Pointer to the index of the last entry enumerated. For the first iteration, initialize this parameter to the constant `expIteratorStart`. Upon return, this references the next entry in the directory. If `pFileInfo` is the last entry, this parameter is set to `expIteratorStop`.

`<- pFileInfo` Pointer to the `FileInfoType` data structure that contains information about the given directory entry.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrEnumerationEmpty
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrFileBadRef
vfsErrInvalidOperation
vfsErrNotADirectory
vfsErrNoFileSystem
```

For more information about the error codes, see “VFS Manager Error Codes” on page 230.

Comments The file system function returns information on the entry referenced by `pui32DirEntryIterator`. The directory to be enumerated must first be opened with `VFSFileOpen` to obtain a file reference number. To get information on all entries in a directory, you must make repeated calls to `VFSDirEntryEnumerate` inside a loop. Boundaries on the iteration are the defined constants `expIteratorStart` and `expIteratorStop`. Before the first call to `VFSDirEntryEnumerate`, initialize `pui32DirEntryIterator` to the constant value `expIteratorStart`. Each iteration then increments the value pointed to by `pui32DirEntryIterator` to the next entry. When this function returns information on the last entry in the directory, `pui32DirEntryIterator` is set to `expIteratorStop`.

IMPORTANT: Creating, renaming, or deleting any file or directory invalidates the enumeration. After any such operation, the enumeration will need to be restarted.

Example The following illustrates how to use `VFSDirEntryEnumerate`.

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

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

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

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileOpen`, "Directory Entry Iterator Start/Stop Constants" on page 164

VFSExportDatabaseToFile

Purpose Flattens and exports the specified database on the handheld to the specified PDB or PRC file on an expansion card.

Declared In `VFSMgr.h`

Prototype

```
long VFSExportDatabaseToFile (WORD volRefNum,
const char *pszPathName, WORD wCardNumber,
LocalID dbID)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume on which to create the destination file.
`-> pszPathName`	Pointer to the full path and filename of the destination file to create. All parts of the path, excluding the filename, must already exist.
`-> wCardNumber`	RAM card number in the handheld on which the database exists. Note that this does not refer to the expansion card and is therefore not related to the slot reference number. The card number for the first RAM memory card on the handheld is 0, which is the only one that most handhelds have.

`-> dbID`	The local ID of the database on the handheld.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
vfsErrBadName
vfsErrInvalidOperation
SYNCERR_REMOTE_BAD_ARG
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments This utility function flattens and exports a database from primary storage memory on a handheld to a PDB or PRC file on an expansion card. This function is the opposite of `VFSImportDatabaseFromFile`. Use this function, for example, to copy applications from primary storage to an expansion card.

Example The following example illustrates how to use `VFSExportDatabaseToFile` to export the MemoPad database.

```
SyncFindDbByTypeCreatorParams rParam;
   SyncDatabaseInfoType rInfo;

   memset (&rParam, 0, sizeof (rParam));
   rParam.bSrchFlags = SYNC_DB_SRCH_OPT_NEW_SEARCH;
   rParam.dwCreator = 'memo';

   SyncFindDbByTypeCreator (rParam, rInfo);
   long retval = VFSExportDatabaseToFile (volRefNum,
      "/Palm/Launcher/Memopad.pdb", 0,
      rInfo.dwLocalId);
```

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileWrite`, `VFSImportDatabaseFromFile`

VFSFileClose

Purpose Closes an opened file or directory.

Declared In `VFSMgr.h`

Prototype `long VFSFileClose (FileRef fileRef)`

Parameters

`-> fileRef`	File reference number passed back from `VFSFileOpen`.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrFileBadRef
vfsErrInvalidOperation
```

For more information about the error codes, see “VFS Manager Error Codes” on page 230.

Comments Use `VFSFileClose` to close a file or directory that has been opened with `VFSFileOpen`.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see “Checking for Expansion Cards” on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileOpen`

VFSFileCreate

Purpose Creates a file given a volume reference number and a path. This function cannot be used to create a directory; use `VFSDirCreate` instead.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileCreate (WORD volRefNum,
const char *pszPathName)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume on which to create the file.
`-> pszPathName`	Pointer to the full path of the file to be created. All parts of the path, excluding the filename, must already exist.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrFileAlreadyExists
vfsErrInvalidOperation
vfsErrNoFileSystem
vfsErrVolumeBadRef
vfsErrVolumeFull
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments All parts of the path except the last component must already exist. `VFSFileCreate` does not open the file. Any operations you want to perform on this directory require a reference, which is obtained through a call to `VFSFileOpen`.

It is the responsibility of the file system library on the handheld to ensure that all filenames are translated into a format that is compatible with the native format of the file system, such as the 8.3 convention for a FAT file system without long filename support. See "Directory Paths" on page 101 in the *C/C++ Sync Suite Companion for Macintosh* for a description of how to construct a valid path.

This function does not open the file. `VFSFileOpen` must be used to open the file. Neither does it create a directory. To create a directory use `VFSDirCreate`.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileDelete`, `VFSFileOpen`, `VFSDirCreate`

VFSFileDelete

Purpose Deletes a closed file or directory.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileDelete (WORD volRefNum,
const char *pszPathName)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume on which to delete the file.
`-> pszPathName`	Pointer to the full path of the file or directory to delete.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrDirNotEmpty
vfsErrFileNotFound
```

```
vfsErrFilePermissionDenied
vfsErrFileStillOpen
vfsErrInvalidOperation
vfsErrNoFileSystem
vfsErrVolumeBadRef
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments A directory must be empty before `VFSFileDelete` can delete it.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileCreate`, `VFSDirCreate`, `VFSFileClose`

VFSFileEOF

Purpose Gets end-of-file status for an open file.

Prototype `long VFSFileEOF (FileRef fileRef)`

Parameters

`-> fileRef`	File reference number passed back by `VFSFileOpen`.

Result If successful, returns 0 (the file pointer was not already at the end of the file).

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrFileBadRef
vfsErrFileEOF
vfsErrInvalidOperation
vfsErrIsADirectory
vfsErrNoFileSystem
```

Comments This function operates only on files and cannot be used with directories.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileOpen`

VFSFileGet

Purpose Reads the file from the expansion card on the handheld and copies it to the desktop.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileGet (WORD volRefNum,
const char *pszDevicePathName,
const char *pszDiskPathName)
```

Parameters

-> `volRefNum` Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume on which the file is present.

-> `pszDevicePathName`
Full path and filename of the file to be read from the expansion card on the handheld.

-> `pszDiskPathName`
Full path and filename for the file to be created on the desktop. All parts of the path, except the file, must already exist. If the file does not exist, then this function creates it. If the file exists, then it overwrites the file.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrDiskFileAccess
vfsErrFileAccessOther
vfsErrFileBadRef
vfsErrFileEOF
vfsErrFilePermissionDenied
vfsErrInvalidOperation
vfsErrIsADirectory
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments Using `VFSFileGet` to copy a file to the desktop is easier than opening the file on the expansion card and reading it into a buffer on the desktop.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileOpen`, `VFSFilePut`

VFSFileGetAttributes

Purpose Gets the attributes of an open file or directory.

Declared In `VFSMgr.h`

Prototype
```
long VFSFileGetAttributes (FileRef fileRef,
UINT32 *pui32Attributes)
```

Parameters

->	`fileRef`	File reference number passed back by `VFSFileOpen`.

`<- pui32Attributes`
Pointer to the attributes of the file or directory. See "File and Directory Attributes" on page 176 for a list of values that can be passed back through this parameter.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrFileBadRef
vfsErrInvalidOperation
vfsErrNoFileSystem
```

Comments The file or directory must be open before calling this function.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileOpen`, `VFSFileGetDate`, `VFSFileSetAttributes`

VFSFileGetDate

Purpose Gets the dates of an open file or directory.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileGetDate (FileRef fileRef,
WORD whichDate, UINT32 *pui32Date)
```

Parameters

`-> fileRef` File reference number passed back by `VFSFileOpen`.

`-> whichDate` Specifies which date — creation, modification, or last access — you want. Supply one of the values described in "Date Type Constants" on page 179.

`<- pui32Date`	Pointer to the requested date. This field is expressed in the standard Palm OS date format — the number of seconds since midnight (00:00:00) January 1, 1904.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
expErrUnsupportedOperation
SYNCERR_REMOTE_BAD_ARG
vfsErrFileBadRef
vfsErrInvalidOperation
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments The file or directory must be open before calling this function. Note that not all file systems are required to support all date types. If the supplied date type is not supported by the file system, `VFSFileGetDate` returns `expErrUnsupportedOperation`.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileOpen`, `VFSFileGetAttributes`, `VFSFileSetDate`

VFSFileOpen

Purpose Opens a file or directory and returns a reference pointer to it.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileOpen (WORD volRefNum,
const char *pszPathName, WORD openMode,
FileRef *pFileRef)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume on which to open the file.
`-> pszPathName`	Pointer to the full path of the file or directory to be opened. This must be a valid path. It cannot be empty and cannot contain null characters. The format of the path should match what the underlying file system supports. See "Directory Paths" on page 101 in the *C/C++ Sync Suite Companion for Macintosh* for a description of how to construct a valid path.
`-> openMode`	Mode to use when opening the file. See "Open Mode Constants" on page 175 for a list of accepted modes.
`<- pFileRef`	Pointer to the opened file or directory.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrFileNotFound
vfsErrFilePermissionDenied
vfsErrInvalidOperation
vfsErrVolumeBadRef
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments The file reference number (`pFileRef`) obtained for a directory cannot be used for all functions. For example, it is not permitted (or logical) to read directly from an opened directory.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileClose`, `VFSDirEntryEnumerate`

VFSFilePut

Purpose Reads a file from the desktop and copies it to an expansion card on the handheld.

Declared In `VFSMgr.h`

Prototype

```
long VFSFilePut (WORD volRefNum,
const char *pszDevicePathName,
const char *pszDiskPathName)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume on which to put the file.
`-> pszDevicePathName`	Full path and filename of the destination file on the handheld. All parts of the path, except the file, must exist. Can also be set to `NULL` (see "Comments" below).
`-> pszDiskPathName`	Full path and filename for the file to be read from the desktop.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrDirectoryNotFound
vfsErrDiskFileAccess
vfsErrFileAccessOther
vfsErrFileAlreadyExists
vfsErrFileNotFound
vfsErrFilePermissionDenied
vfsErrInvalidOperation
vfsErrNoFileSystem
vfsErrVolumeBadRef
vfsErrVolumeFull
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments The behavior of this function depends on whether a destination path is specified:

- If `pszDevicePathName` is specified, all parts of the path, except the filename, must already exist.
  - If the full path exists, this function copies the file to specified location.
  - If the full path does *not* exist, this function fails and returns an error.
- If `pszDevicePathName` is `NULL` or points to an empty string:
  - If a default directory is registered for this file type, the VFS Manager ensures that the entire path exists — creating the directories leading up to the default directory, if necessary — and puts the file in the default directory.
  - If no default directory is registered for this file type, this function returns `vfsErrDirectoryNotFound`.

If the path exists in either of the above cases, this function copies the file specified by `pszDiskPathName` to the destination on the expansion card. If the file already exists at the destination, this function overwrites it with the one specified by `pszDiskPathName`.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSGetDefaultDirectory`, `VFSFileGet`

VFSFileRead

Purpose Reads data from a file into the specified buffer.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileRead (FileRef fileRef,
UINT32 numBytes, void *pBuffer,
UINT32 *pNumBytesRead)
```

Parameters

`-> fileRef`	File reference number passed back by `VFSFileOpen`.
`-> numBytes`	Number of bytes to read.
`<- pBuffer`	A pointer to the destination memory chunk on the desktop where the data is stored. The caller must preallocate this buffer to hold at least `numBytes` characters.
`<- pNumBytesRead`	A pointer to an unsigned integer that reflects the number of bytes actually read. This value is set on return and does not need to be initialized. If no bytes are read, the value is set to zero.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrFileBadRef
vfsErrFileEOF
vfsErrFilePermissionDenied
vfsErrInvalidOperation
vfsErrIsADirectory
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments This function operates only on files and cannot be used with directories; use `VFSDirEntryEnumerate` to explore the contents of a directory.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileWrite`, `VFSImportDatabaseFromFile`

VFSFileRename

Purpose Renames a closed file or directory.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileRename (WORD volRefNum,
const char *pszPathName, const char *pszNewName)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume on which to rename the file.
`-> pszPathName`	Pointer to the full path of the file or directory to rename.

`-> pszNewName`	Pointer to the new filename. Note that this is the name of the file only and does not include the path to the file.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrFileAlreadyExists
vfsErrFileNotFound
vfsErrFilePermissionDenied
vfsErrFileStillOpen
vfsErrInvalidOperation
vfsErrNoFileSystem
vfsErrVolumeBadRef
vfsErrVolumeFull
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments This function cannot be used to move a file to another location within the file system. This function returns `vfsErrBadName` if either `pszPathName` or `pszNewName` is invalid, or if the string pointed to by `pszNewName` is a path rather than a filename.

Example Below is an example of how to use `VFSFileRename`. Note that the renamed file remains in the `/PALM/Programs` directory; `VFSFileRename` can't be used to move files from one directory to another.

```
// volRefNum must have been previously defined; most likely,
// it was returned by VFSVolumeEnumerate.

err = VFSFileRename (volRefNum, "/PALM/Programs/foo.prc",
                     "bar.prc");
if (err != 0) {
    // Handle error...
}
```

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileCreate`, `VFSFileDelete`

VFSFileResize

Purpose Changes the size of an open file. This function operates only on files and cannot be used with directories.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileResize (FileRef fileRef,
UINT32 ui32NewSize)
```

Parameters

`-> fileRef`	File reference number passed back from `VFSFileOpen`.
`-> ui32NewSize`	The desired new size of the file. This can be larger or smaller than the current file size.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrFileBadRef
vfsErrInvalidOperation
vfsErrIsADirectory
vfsErrNoFileSystem
vfsErrVolumeFull
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments If the resizing of the file would make the current file pointer point beyond the end of the file, this function sets the file pointer to the end of the file.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileSize`

VFSFileSetAttributes

Purpose Sets the attributes of an open file or directory.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileSetAttributes (FileRef fileRef,
UINT32 ui32Attributes)
```

Parameters

`-> fileRef`	File reference number passed back from `VFSFileOpen`.
`-> ui32Attributes`	Attributes to associate with the file or directory. See "File and Directory Attributes" on page 176 for a list of values you can use when setting this parameter.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrFileBadRef
vfsErrInvalidOperation
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments You cannot use this function to set the `vfsFileAttrDirectory` or `vfsFileAttrVolumeLabel` attributes. The `vfsFileAttrDirectory` is set when you call `VFSDirCreate`. The `vfsFileAttrVolumeLabel` is set when you call `VFSVolumeSetLabel`.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileGetAttributes`, `VFSFileSetDate`

VFSFileSetDate

Purpose Changes the dates of an open file or directory.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileSetDate (FileRef fileRef,
UINT32 whichDate, UINT32 date)
```

Parameters

`-> fileRef`	File reference number passed back in `VFSFileOpen`.
`-> whichDate`	Specifies which date — creation, modification, or last access — to modify. Supply one of the Date Type Constants in this parameter. Note that not all file systems are required to support all date types. If the supplied date type is not supported by the file system, `VFSFileGetDate` returns `expErrUnsupportedOperation`.
`-> date`	The new date. Express this parameter in the standard Palm OS date format — the number of seconds since midnight (00:00:00) January 1, 1904.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
expErrUnsupportedOperation
SYNCERR_REMOTE_BAD_ARG
vfsErrFileBadRef
vfsErrFilePermissionDenied
vfsErrInvalidOperation
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments If the `whichDate` parameter is not one of the defined date type constants, this function returns `SYNCERR_REMOTE_BAD_ARG`. However, if `whichDate` is one of the defined constants but is not one supported by the file system, this function returns `expErrUnsupportedOperation`.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileGetDate`, `VFSFileSetAttributes`

VFSFileSeek

Purpose Sets the position from which to read or write within an open file. This function operates only on files and cannot be used with directories.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileSeek (FileRef fileRef,
FileOrigin origin, Int32 offset)
```

Parameters	`-> fileRef`	File reference number passed back from `VFSFileOpen`.
	`-> origin`	Origin to use when calculating the new position. The `offset` parameter indicates the desired new position relative to this origin, which must be one of the values defined in "File Origin Constants" on page 179.
	`-> offset`	Offset, either positive or negative, from the origin to which to set the current position. A value of zero positions you at the specified origin.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrFileBadRef
vfsErrFileEOF
vfsErrInvalidOperation
vfsErrIsADirectory
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments If the resulting position of the file pointer would be beyond the end of the file, this function sets the position to the end of the file. Similarly, if the resulting position of the file pointer would be before the beginning of the file, this function sets the position to the beginning of the file.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileSize`, `VFSFileTell`

VFSFileSize

Purpose Gets the size of an open file.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileSize (FileRef fileRef,
UINT32 *pui32FileSize)
```

Parameters

`-> fileRef`	File reference number passed back from `VFSFileOpen`.
`<- pui32FileSize`	Pointer to the size of the open file.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrFileBadRef
vfsErrInvalidOperation
vfsErrIsADirectory
vfsErrNoFileSystem
```

For more information about the error codes, see “VFS Manager Error Codes” on page 230.

Comments This function operates only on files and cannot be used with directories.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see “Checking for Expansion Cards” on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileResize`, `VFSFileTell`, `VFSVolumeSize`

VFSFileTell

Purpose Gets the current position of the file pointer within an open file.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileTell (FileRef fileRef,
UINT32 *pFilePos)
```

Parameters

`-> fileRef`	File reference number passed back from `VFSFileOpen`.
`<- pFilePos`	Pointer to the current position of the file pointer.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrFileBadRef
vfsErrInvalidOperation
vfsErrIsADirectory
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments This function operates only on files and cannot be used with directories.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSFileSeek`, `VFSFileSize`

VFSFileWrite

Purpose Writes data to an open file.

Declared In `VFSMgr.h`

Prototype

```
long VFSFileWrite (FileRef fileRef,
UINT32 numBytes, const void *pDataBuf,
UINT32 *pNumBytesWritten)
```

Parameters

`-> fileRef`	File reference number passed back from `VFSFileOpen`.
`-> numBytes`	The number of bytes to write.
`-> pDataBuf`	A pointer to a buffer containing the data to write.
`<- pNumBytesWritten`	A pointer to an unsigned integer that reflects the number of bytes actually written. This value is set on return and does not need to be initialized. If no bytes are written the value is set to zero.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrFileBadRef
vfsErrFilePermissionDenied
vfsErrInvalidOperation
vfsErrIsADirectory
vfsErrNoFileSystem
vfsErrVolumeFull
```

For more information about the error codes, see “VFS Manager Error Codes” on page 230.

Comments This function operates only on files and cannot be used with directories.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSExportDatabaseToFile`, `VFSFileRead`

VFSGetAPIVersion

Purpose Retrieves the version of the Expansion Manager and VFS Manager APIs.

Declared In `VFSMgr.h`

Prototype
```
long VFSGetAPIVersion (DWORD *pdwMajor,
DWORD *pdwMinor)
```

Parameters

`<- pdwMajor`	Pointer to variable for returning the *major* version number; pass `NULL` to ignore.
`<- pdwMinor`	Pointer to variable for returning the *minor* version number; pass `NULL` to ignore.

Result Always returns 0.

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments The Expansion Manager and VFS Manager APIs strive to maintain backward compatibility within a given *major* version number group. As new exported functions are added to the API s or critical bugs are fixed, the *minor* version number of the APIs will be incremented and the documentation of the new functions will identify the APIs version number where they were first available. Conduits can check the version number using the `VFSGetAPIVersion` function. For example, if a conduit requires a bug fix for a particular VFS Manager function that was made in VFS Manager API version number 2.1, the conduit must call `VFSGetAPIVersion` to make sure that the *major* number is 2 and the *minor* number is 1 or greater before making calls to the new function.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

VFSGetDefaultDirectory

Purpose Retrieves the default directory on the given volume for files of a particular type.

Declared In `VFSMgr.h`

Prototype

```
long VFSGetDefaultDirectory (WORD volRefNum,
const char *pszFileType, char *pszDirPath,
WORD *pwPathLen)
```

Parameters

`-> volRefNum`	Volume reference number passed back by `VFSVolumeEnumerate`.
`-> pszFileType`	A pointer to the requested file type, as a `NULL`-terminated string. The file type may either be a MIME media type/subtype pair, such as "image/jpeg", "text/plain", or "audio/basic"; or a file extension, such as ".jpeg". If you pass in a file extension, it must begin with a period '`.`' — for example "`.prc`".
`<- pszDirPath`	A pointer to the buffer that receives the default directory path for the requested file type. The caller must allocate this buffer before calling this function.
`<-> pwPathLen`	A pointer to the size of the path. On entry, set this to the size of `pszDirPath` buffer. On return, reflects the number of bytes copied to `pszDirPath`.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrBufferOverflow
vfsErrFileNotFound
vfsErrInvalidOperation
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments This function returns the complete path to the default directory registered for the specified file type. A default directory can be registered for each type of media supported. The directory should be registered under media and file type. (Note that this directory is typically a "root" directory for the file type; any subdirectories under this root directory should also be searched for files of the appropriate type.) If this function finds no match for either the specified media type for this volume or the requested file type, it returns `vfsErrFileNotFound`.

If a match is found, but the `pszDirPath` buffer is too small to hold the resulting path string, this function returns `vfsErrBufferOverflow`.

This function can be used by an image viewer application, for example, to find the directory containing images without having to know what type of media the volume was on. This could be "`/DCIM`", "`/images`", or something else depending on the type of media.

Example This example illustrates how to use the `VFSGetDefaultDirectory` function.

```
char fileTypeStr [] = ".prc";
char devicePathBuffer [MAX_PATH];
WORD bufLen = sizeof (devicePathBuffer);

long retval = VFSGetDefaultDirectory
                (volRefNum, fileTypeStr, devicePathBuffer,
                 &bufLen);
if (0 == retval)
```

```
{
  // Got the default directory.
  // Perform further operations here.
}
else
{
  // Could not get the default path on the card:
  // process error codes.
}
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSDirEntryEnumerate`

VFSImportDatabaseFromFile

Purpose Creates a database from the specified `.pdb` or `.prc` file on an expansion card.

Declared In `VFSMgr.h`

Prototype
```
long VFSImportDatabaseFromFile (WORD volRefNum,
const char *pszPathName, WORD *cardNoP,
LocalID *dbIDP)
```

Parameters

-> `volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume from which to get the source file.
-> `pszPathName`	A pointer to the full path and name of the source file to get.

<- `cardNoP`	A pointer to a variable that receives the RAM card number of the newly-created database. If the database already resides in the storage heap, the card number of the existing database is passed back and the error `SYNCERR_FILE_ALREADY_EXIST` is returned.
<- `dbIDP`	A pointer to a variable that receives the database ID of the new database. If the database already resides in the storage heap, the database ID of the existing database is passed back and the error `SYNCERR_FILE_ALREADY_EXIST` is returned.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
SYNCERR_FILE_ALREADY_EXIST
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrInvalidOperation
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments This function imports a `.pdb` or `.prc` file on an expansion card into a new database in the handheld storage heap. If the database already exists, this function passes back values in `cardNoP` and `dbIDP` for the existing database and returns an error code of `SYNCERR_FILE_ALREADY_EXIST`.

This function is used, for example, to copy applications from a storage card to main memory.

Example This example illustrates the use of the `VFSImportDatabaseFromFile` function.

```
long retval = VFSImportDatabaseFromFile
    (volRefNum, "/Palm/Launcher/Contacts.pdb",
     &cardNo, &dBID);
```

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSExportDatabaseToFile`, `VFSFileRead`

VFSSupport

Purpose Determines whether the Expansion Manager is present on the handheld, its version if present, and gets expansion slot and volume information.

Declared In `VFSMgr.h`

Prototype

```
long VFSSupport (DWORD &dwExpansionMgrVersion,
DWORD &dwVolumesAvailable)
```

Parameters

`<- dwExpansionMgrVersion`
When this parameter passes back a zero value, no expansion slot is present. A nonzero value is the version of Expansion Manager on the handheld.

`<- dwVolumesAvailable`
When this parameter passes back a zero value, either no file system is present on the card in the slot or no card is in the slot. A nonzero value is the number of volumes present on the card.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

`SYNCERR_BAD_ARG`

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments This information has already been obtained by the desktop VFS Manager, so no additional calls are made to the handheld at the time you call this function. If either pointer is `NULL`, this function returns `SYNCERR_BAD_ARG`.

To get the version of the VFS Manager on the handheld, see "Using the SyncReadFeature Function" on page 89 in the *C/C++ Sync Suite Companion for Macintosh*.

Compatibility VFS Manager version: All.
Palm OS version: 4.0.

See Also `VFSGetAPIVersion`, "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*

VFSVolumeEnumerate

Purpose Enumerates the mounted volumes and retrieves a list of volume reference numbers.

Declared In `VFSMgr.h`

Prototype

```
long VFSVolumeEnumerate (WORD *pwNumVolumes,
WORD *pwVolRefList)
```

Parameters

`<- pwNumVolumes`
A pointer to the number of volumes successfully enumerated.

`<-> pwVolRefList`
On exit, a pointer to an array of volume reference numbers. If the caller passes in `NULL`, the function passes back no volume reference numbers. If `NULL` is not passed in, the caller must allocate sufficient space to hold all volume reference numbers.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrEnumerationEmpty
vfsErrInvalidOperation
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments This function passes back a list of reference numbers of all of the volumes that are mounted. The list can span across expansion cards, if multiple cards are present. To find which card and slot this volume is mounted from, call `VFSVolumeInfo`.

Before calling `VFSVolumeEnumerate` to get volume reference numbers, the caller must allocate enough space for the array pointed to by `pwVolRefList`. If the caller passes in `NULL` for `pwVolRefList`, the function returns only the number of volumes. Use this to allocate the array and call `VFSVolumeEnumerate` again to get the volume reference numbers.

Example The following example shows how to use `VFSVolumeEnumerate` to get the number of mounted volumes, allocate a buffer, and get the list of volume reference numbers.

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

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

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSVolumeInfo`

VFSVolumeFormat

Purpose Formats and mounts the first volume installed in a given slot.

Declared In `VFSMgr.h`

Prototype

```
long VFSVolumeFormat (BYTE byMountFlags,
WORD fsLibRefNum,
VFSAnyMountParamPtr pVfsMountParam,
WORD vfsMountParamLen)
```

Parameters

`-> byMountFlags`
This parameter is reserved. Pass in zero.

`-> fsLibRefNum` This parameter is reserved. Pass in zero.

`<-> pVfsMountParam`
Parameters to be used when mounting the volume after it has been formatted. Supply a pointer to a `VFSAnyMountParamType` structure. Note that you must pass in a pointer to a different structure type depending on the value of `pVfsMountParam->mountClass`. For example, if `mountClass` is set to `vfsMountClass_SlotDriver`, then the `pVfsMountParam` you pass in must point to a `VFSSlotMountParamType` structure. Upon exit, this points to a structure of the same type containing a new volume reference number.

`-> vfsMountParamLen`
The length in bytes of the structure passed via `pVfsMountParam`.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotEnoughPower
expErrNotOpen
vfsErrInvalidOperation
vfsErrNoFileSystem
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments `VFSVolumeFormat` attempts to find a compatible file system library on the handheld to mount the volume. A volume can be either mounted or unmounted at the time you call this function. `VFSVolumeFormat` also remounts the volume if the format succeeds. (The handheld slot driver provided by PalmSource, Inc. currently supports only one volume per slot.)

NOTE: For a card that has not been previously formatted (and therefore does not have a volume), set the `volRefNum` member of the `VFSAnyMountParamType` structure to `vfsInvalidVolRef`. Upon exit, this function passes back a valid volume reference number in a structure of the same type.

To use `VFSVolumeFormat` with a file system based on a slot driver:

- The `pVfsMountParam` parameter must point to a `VFSSlotMountParamType` structure.
- The `vfsMountParam.mountClass` member must be set to `vfsMountClass_SlotDriver`.
- The `vfsMountParam.slotLibRefNum` member may be set to `vfsInvalidSlotLibRefNum` (which causes the handheld HotSync® client to look up the proper driver) or to the value obtained for the `VolumeInfoType` structure by calling `VFSVolumeInfo`.

Example The following code excerpt formats a volume on a physical slot using a compatible file system.

```
VFSSlotMountParamType stSlotMountParam;

stSlotMountParam.vfsMountParam.volRefNum = volRefNum; // Or vfsInvalidVolRef if
                                                      // the card has not been
                                                      // formatted.
stSlotMountParam.vfsMountParam.reserved = 0;
stSlotMountParam.vfsMountParam.mountClass = vfsMountClass_SlotDriver;
stSlotMountParam.slotLibRefNum = vfsInvalidSlotLibRefNum;
stSlotMountParam. slotRefNum = slotRefNum;
// We get this from ExpSlotEnumerate.

long retval = VFSVolumeFormat (0, 0, &stSlotMountParam,
  sizeof(stSlotMountParam));
```

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSVolumeInfo`

VFSVolumeGetLabel

Purpose Gets the volume label for a particular volume.

Declared In `VFSMgr.h`

Prototype `long VFSVolumeGetLabel (WORD volRefNum, char *pszVolLabel, WORD *pwBufLen)`

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume for which to get the label.
`<- pszVolLabel`	A pointer to a character buffer that receives the volume name upon return.
`-> pwBufLen`	A pointer to the length, in bytes, of the `pszVolLabel` buffer.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrBufferOverflow
vfsErrInvalidOperation
vfsErrNameShortened
vfsErrNoFileSystem
vfsErrVolumeBadRef
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Comments Volume reference numbers can change each time the handheld mounts a given volume. To keep track of a particular volume, save the volume's label rather than its reference number. Volume labels can be up to 255 characters long. They can contain any normal character, including spaces and lowercase characters, in any character set as well as the following special characters: $ % ' - _ @ ~ ` ! ( ) ^ # & + , ; = [ ].

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSVolumeSetLabel`

VFSVolumeInfo

Purpose Gets information about the specified volume.

Declared In `VFSMgr.h`

Prototype

```
long VFSVolumeInfo (WORD volRefNum,
VolumeInfoType *pVolInfo)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume for which to get information.
`<- pVolInfo`	Pointer to the structure that receives the volume information for the specified volume . See `VolumeInfoType` for more information on the fields in this data structure.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrInvalidOperation
vfsErrNoFileSystem
vfsErrVolumeBadRef
```

For more information about the error codes, see “VFS Manager Error Codes” on page 230.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see “Checking for Expansion Cards” on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSVolumeGetLabel`, `VFSVolumeSize`

VFSVolumeSetLabel

Purpose Changes the volume label for a mounted volume.

Declared In `VFSMgr.h`

Prototype
```
long VFSVolumeSetLabel (WORD volRefNum,
const char *pszVolLabel)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume for which to set the label.
`-> pszVolLabel`	Pointer to the label to apply to the specified volume. This string must be `NULL`-terminated.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
SYNCERR_REMOTE_BAD_ARG
vfsErrBadName
vfsErrInvalidOperation
vfsErrNameShortened
vfsErrNoFileSystem
vfsErrVolumeBadRef
```

For more information about the error codes, see “VFS Manager Error Codes” on page 230.

Comments Volume labels can be up to 255 characters long. They can contain any normal character, including spaces and lowercase characters, in any character set as well as the following special characters: $ % ' - _ @ ~ ` ! ( ) ^ # & + , ; = [ ]. See “Naming Volumes” on page 96 in the *C/C++ Sync Suite Companion for Macintosh* for guidelines on naming.

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

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSVolumeGetLabel`

VFSVolumeSize

Purpose Determines the total amount of space on a volume, as well as the amount that is currently being used.

Declared In `VFSMgr.h`

Prototype

```
long VFSVolumeSize (WORD volRefNum,
UINT32 *pui32UsedSize, UINT32 *pui32TotalCapacity)
```

Parameters

`-> volRefNum`	Volume reference number (passed back by `VFSVolumeEnumerate`) of the volume for which to get the size.
`<- pui32UsedSize`	A pointer to a variable that receives the amount of space, in bytes, in use on the volume.
`<- pui32TotalCapacity`	A pointer to a variable that receives the total capacity of the volume, in bytes.

Result If successful, returns 0.

If unsuccessful, returns one of the following error codes:

```
expErrNotOpen
vfsErrInvalidOperation
vfsErrNoFileSystem
vfsErrVolumeBadRef
```

For more information about the error codes, see "VFS Manager Error Codes" on page 230.

Compatibility VFS Manager version: All.
Palm OS® version: 4.0. Implemented only if the VFS Manager is present on the handheld (see "Checking for Expansion Cards" on page 87 in the *C/C++ Sync Suite Companion for Macintosh*).

See Also `VFSVolumeInfo`

VFS Manager Error Codes

Table 5.9 describes the error codes that VFS Manager functions can return and are defined in `VFSErr.h`. Several Expansion Manager error codes (`exp*`) and Sync Manager error codes (`SYNC*`) can also be returned (see "Expansion Manager Error Codes" on page 171 and "Sync Manager Errors" on page 151).

Table 5.9 VFS Manager error codes

Constant	Description
`vfsErrBadData`	The operation could not be completed because of invalid data — for example, importing a database from a corrupted `.prc` file.
`vfsErrBadName`	Invalid filename, path, or volume label. See "Naming Files" on page 99, "Directory Paths" on page 101, or "Naming Volumes" on page 96 in the *C/C++ Sync Suite Companion for Macintosh*.
`vfsErrBufferOverflow`	The supplied buffer is too small.
`vfsErrDirectoryNotFound`	The path, excluding filename, does not exist or no default directory is registered for this file type.
`vfsErrDirNotEmpty`	The directory is not empty and therefore cannot be deleted.
`vfsErrDiskFileAccess`	Failed to create or open the disk file on the desktop.
`vfsErrDiskFull`	Not enough space on the desktop's disk.
`vfsErrFileAccessOther`	Generic desktop file access error. If returned by `VFSFileGet`, could not access or map the desktop file — for example, because of insufficient memory on the desktop.
`vfsErrFileAlreadyExists`	A file or a directory with this name exists in this location already.
`vfsErrFileBadRef`	The file reference number is invalid: it has been closed or was not obtained from `VFSFileOpen`.

Table 5.9 VFS Manager error codes *(continued)*

Constant	Description
`vfsErrFileEOF`	The file pointer has been moved to the end of the file. This code is not considered an error.
`vfsErrFileGeneric`	Generic file error.
`vfsErrFileNotFound`	The file was not found in the specified path.
`vfsErrFilePermissionDenied`	Permission denied to perform requested operation — for example, an attempt to write to a read-only file or to read a file already opened in the `vfsModeExclusive` mode.
`vfsErrFileStillOpen`	The file is still open — for example, trying to delete an open file.
`vfsErrInvalidOperation`	A file system is not present or the VFS Manager function is not valid.
`vfsErrIsADirectory`	This operation can be performed only on a regular file, not a directory.
`vfsErrNameShortened`	A volume name or filename was automatically shortened to conform to the file system specification.
`vfsErrNoFileSystem`	None of the file systems installed on the handheld support this operation.
`vfsErrNotADirectory`	This operation can be performed only on a directory.
`vfsErrUnimplemented`	This call is not implemented.
`vfsErrVolumeBadRef`	The volume reference number is invalid.
`vfsErrVolumeFull`	There is insufficient space left on the volume.
`vfsErrVolumeStillMounted`	The volume is still mounted.

6

User Manager API

Use the User Manager API to access the *users data store* on the desktop computer. The users data store stores name, synchronization preferences, directory, and password information about users who have synchronized Palm Powered™ handhelds with the desktop computer.

The User Manager functions are available in `HotSync Libraries` and declared in `UserMgr.h`.

The sections in this chapter are:

- User Manager API Versions
- User Manager Constants
- User Manager Functions
- User Manager Error Codes

NOTE: The User Manager API is functionally similar to an API that exists for Windows, where is it named the “User Data API.” This API includes Install Aide functionality, which has been depreciated in the Macintosh CDK.

User Manager API Versions

The User Manager API is new to the Mac OS CDK as of version 4.03. Each version of the User Manager API has a major version number and a minor version number. You can determine the version of the User Manager API that you are using by calling the `UmGetLibVersion` function.

The User Manager API maintains backward compatibility within a major version. The minor version number changes when new functions are added or bugs are fixed. This document includes version information for each function.

NOTE: Your application must check for the presence of the User Manager API by checking for the existence of the `UmGetLibVersion` function prior to issuing any User Manager calls.

If your application depends on functions that are available only in certain versions of the User Manager API, you need to determine the version of the User Manager API with which you are dealing on a specific installation. To do so, call the `UmGetLibVersion` function, which returns both the major version number and minor version number of the User Manager API on the desktop computer.

For example, if your application depends on a function that was added in version 2.0 of the User Manager API, you need to call the `UmGetLibVersion` function and then verify that the major number is `2` or greater and that the minor version number is `0` or greater.

NOTE: This reference documents version 1.0 of the User Manager API.

User Manager Constants

Current User ID (kCurrentPalmUserID) Constant

In many cases in which you need to specify a Palm user ID, you can use the `kCurrentPalmUserID` constant instead.

```
enum {
                        kCurrentPalmUserID= 0;
};
```

User Synchronization Action (UmUserSyncAction) Constants

The user synchronization action constants specify a user's preferences for the type of synchronization operation to perform for a specified conduit.

```
typedef UInt16 UmUserSyncAction;
enum {
                       kUmSynchronize = 0,
                       kUmPCToHH = 1,
                       kUmHHToPC = 2,
                       kUmDoNothing = 3,
                       kUmProfileInstall = 4,
                       kUmCustom = 5,
                       kUmInstall = 6,
                       kUmBackup = 7
};
```

`kUmSynchronize`	Perform a mirror-image synchronization between the desktop computer and the handheld.
`kUmPCToHH`	Perform a restore from the desktop computer: overwrite the database on the handheld with the database on the desktop computer.
`kUmHHToPC`	Perform a restore from the handheld: overwrite the desktop database with the database on the handheld.
`kUmDoNothing`	Do not exchange data between the handheld and the desktop computer; the conduit does, however, load and can set flags or log messages.
`kUmProfileInstall`	Perform a profile download. A profile is a special user account that you can set up on the desktop computer that downloads data to a handheld, erasing all information on the handheld and leaving it without a user ID.
`kUmCustom`	Perform any custom actions implemented in the conduit. HotSync Manager passes only this flag to the conduit, which must determine what action to take.
`kUmInstall`	Install new applications from the desktop computer to the handheld.

`kUmBackup`	Perform a backup of the databases on the handheld to the desktop computer.

User Manager Functions

This section describes the following User Manager functions for application use:

- `UmAddUser`
- `UmCopyRootDirectory`
- `UmCopySlotName`
- `UmCopyUserDirectory`
- `UmDeleteKey`
- `UmDeleteUser`
- `UmDeleteUserPermSyncPreferences`
- `UmDeleteUserTempSyncPreferences`
- `UmGetCurrentUser`
- `UmGetFilesToInstallFolderSpec`
- `UmGetGlobalConduitsDirectory`
- `UmGetIDFromDirectory`
- `UmGetIDFromName`
- `UmGetInteger`
- `UmGetLibVersion`
- `UmGetSlotCount`
- `UmGetSlotFolderSpec`
- `UmGetString`
- `UmGetUserByDirName`
- `UmGetUserCount`
- `UmGetUserDataLastModDate`
- `UmGetUserID`
- `UmGetUserName`
- `UmGetUserNameByID`

- UmGetUserPassword
- UmGetUserPermSyncPreferences
- UmGetUserTempSyncPreferences
- UmGetUsersConduitsDirectory
- UmIsUserInstalled
- UmIsUserProfile
- UmRemoveUserTempSyncPreferences
- UmSetInteger
- UmSetString
- UmSetUserDirectory
- UmSetUserInstall
- UmSetUserNameByID
- UmSetUserPermSyncPreferences
- UmSetUserTempSyncPreferences

New

UmAddUser

Purpose Adds a user to the users data store. If the users data store does not exist, this function creates a new one and adds the user to it.

Prototype

```
OSStatus UmAddUser (CFStringRef iUserName,
Boolean iIsProfileUser);
```

Parameters

`-> iUserName`	A pointer to a character buffer that specifies the user to add.
`-> iIsProfileUser`	If this is `true`, the new user is to be a profile user; if this is `false`, the new user is not to be a profile user.

Result If successful, returns `kUserMgrNoErr`. If the users data store does not exist, this function creates a new one, adds the user, and returns `kUserMgrNoErr` if successful.

If unsuccessful, returns one of the following error code values:

```
kUserMgrParamErr
kUserMgrCorruptUsersFileErr
kUserMgrInvalidUserNameErr
kUserMgrSaveErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmIsUserProfile`, `UmDeleteUser`

New UmCopyRootDirectory

Purpose Retrieves a `CFURLRef` to the directory containing all of the user data directories.

Prototype

```
OSStatus UmCopyRootDirectory
(CFURLRef *oRootUserDirectory);
```

Parameters `<- oRootUserDirectory`
A pointer to a `CFURLRef` buffer in which to pass back the path reference.

Result If successful, returns `kUserMgrNoErr` and places a reference to the root directory URL in `oRootUserDirectory`.

If unsuccessful, returns one of the following error codes.

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments `UmCopyRootDirectory` retrieves the root directory in which user directories are stored, such as `Macintosh HD:Documents:Palm:Users`. Therefore, to get the complete path of a specific user directory, concatenate the results of `UmCopyRootDirectory` and `UmCopyUserDirectory`— for example, `Macintosh HD:Documents:Palm:Users:<user_directory>`.

Compatibility User Manager version: All.

See Also `UmCopyUserDirectory`, `UmSetUserDirectory`

New UmCopySlotName

Purpose Retrieves the slot name for the given slot on the specified user's handheld.

Prototype

```
OSStatus UmCopySlotName (PalmUserID iUserID,
UInt16 iSlot, CFStringRef *oSlotName);
```

Parameters

`-> iUserID`	The ID of the user. `kCurrentPalmUserID` is allowed.
`-> iSlot`	The ID of the slot for which to get the name.
`<- oSlotName`	Pointer to a buffer to receive a `CFStringRef` indicating the slot's name.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrInvalidSlotIndexErr
```

This function can also return File Manager errors.

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments Use the slot name to identify the slot for the user's benefit, not the slot ID.

HotSync Manager assigns names to slots based on their media type at the beginning of each HotSync operation and saves it for the corresponding user in the user information store on the desktop. This function simply passes back the saved information. Therefore it may not be accurate for the next HotSync operation, because the user may have changed or updated the handheld.

Compatibility User Manager version: All.
Palm OS® version: 4.0 or later.

New UmCopyUserDirectory

Purpose Returns a URL reference to the directory.

Prototype

```
OSStatus UmCopyUserDirectory (PalmUserID iUserID,
CFURLRef *oUserDirectory);
```

Parameters

`-> iUserID`	The user ID of the user whose user data directory is to be returned. You may specify `kCurrentPalmUserID`.
`-> oUserDirectory`	On return, contains a `CFURLRef` reference to the user's data directory.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrNoDirectoryErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmSetUserDirectory`

New UmDeleteKey

Purpose Deletes a key or an entire section from the specified user's area of the users data store.

Prototype

```
OSStatus UmDeleteKey (PalmUserID iUserID,
CFStringRef iSectionName, CFStringRef iKeyName);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iSectionName`	The section name in the specified user's area of the users data store.
`-> iKeyName`	The key to delete. If this is `NULL`, then the entire section is deleted.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values.

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
kUserMgrParamErr
```

This function can return File Manager errors.

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`

New UmDeleteUser

Purpose Deletes a user from the users data store.

Prototype `OSStatus UmDeleteUser (PalmUserID iUserID);`

Parameters	`-> iUserID`	The user ID, which specifies the user to reference in the users data store.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
kUserMgrSaveErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmAddUser`

New UmDeleteUserPermSyncPreferences

Purpose Deletes the permanent synchronization preferences for *all* of the specified user's conduits.

Prototype
```
OSStatus UmDeleteUserPermSyncPreferences
(PalmUserID iUserID);
```

Parameters	`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments The `UmDeleteUserPermSyncPreferences` function unsets the temporary synchronization preference. The result is the same as if the user has never clicked HotSync Manager's **Custom** > **Change** option and altered a synchronization preference.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetUserPermSyncPreferences`, `UmGetUserPermSyncPreferences`

New UmDeleteUserTempSyncPreferences

Purpose Deletes the temporary synchronization preferences for *all* of the specified user's conduits.

Prototype

```
OSStatus
UmDeleteUserTempSyncPreferences(PalmUserID iUserID
);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments The `UmDeleteUserPermSyncPreferences` function unsets the temporary synchronization preference. The result is the same as if the user has never clicked HotSync Manager's **Custom** > **Change** option and altered a synchronization preference.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetUserTempSyncPreferences`, `UmGetUserTempSyncPreferences`, `UmRemoveUserTempSyncPreferences`

New UmGetCurrentUser

Purpose Returns the user ID of the current user.

Prototype

```
OSStatus UmGetCurrentUser (PalmUserID *oUserID);
```

Parameters

`<- oUserID`	A pointer to a `PalmUserID` buffer. On return, this buffer contains the ID of the current handheld user.

Result If successful, stores the current handheld user's ID in `oUserID`, and returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

New UmGetFilesToInstallFolderSpec

Purpose Returns an `FSSpec` reference to the "Files to Install" folder for the specified user.

Prototype

```
OSStatus UmGetFilesToInstallFolderSpec
(PalmUserID iUserID, FSSpec *oFilesToInstallSpec);
```

Parameters

`-> iUserID` The User ID of the user whose "Files to Install" folder is to be located. You may specify `kCurrentPalmUserID`.

`<- oFilesToInstallSpec` A buffer to receive the `FSSpec` of the "Files to Install" folder.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

New

UmGetGlobalConduitsDirectory

Purpose Returns the `VRefNum` and `DirID` of the global conduits directory.

Prototype

```
OSStatus UmGetGlobalConduitsDirectory (SInt16
iDomain, SInt16 *oConduitsVRefNum, SInt32
*oConduitsDirID);
```

Parameters

`-> iDomain`	The Folder Domain to look in. See Chapter 9, "Installing Conduits," on page 111 in the *C++ Sync Suite Companion for Macintosh* for details.
`<- oConduitsVRefNum`	A buffer to receive the volume reference number.
`<- oConduitsDirID`	A buffer to receive the directory ID number.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

`kUserMgrParamErr`

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments This function takes the place of hunting down the active Serial Monitor in the desktop database and finding its directory.

Compatibility User Manager version: All.

New UmGetIDFromDirectory

Purpose Retrieves a user ID given the user directory.

Prototype

```
OSStatus UmGetIDFromDirectory
(CFURLRef iUserDirectory, PalmUserID *oUserID);
```

Parameters

`-> iUserDirectory`	A `CFURLRef` that contains the path to the user directory.
`<- oUserID`	A buffer to receive the user ID.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorrrruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`

New UmGetIDFromName

Purpose Retrieves a user ID given the user's name.

Prototype

```
OSStatus UmGetIDFromName (CFStringRef iUserName,
PalmUserID *oUserID);
```

Parameters

`-> iUserName`	A `CFStringRef` indicating the user's name.
`<- oUserID`	A buffer to receive the user ID.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrInvalidUserNameErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments Note that it is possible for the users data store to contain the same name more than once. Because the user ID is the only value that the User Manager API ensures is unique, each instance of the same name has a different user ID. Therefore, in that case, you must perform additional checking to determine whether the user name is unique.

Compatibility User Manager version: All.

See Also `UmGetUserID`

New UmGetInteger

Purpose Retrieves an integer value from a key in the specified user's area of the users data store.

Prototype

```
OSStatus UmGetInteger (PalmUserID iUserID,
CFStringRef *iSectionName, CFStringRef *iKeyName,
SInt32 iDefaultValue, SInt32 *oValue);
```

Parameters

`-> iUserID` The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.

`-> iSectionName`
The section name in the specified user's area of the users data store.

`-> iKeyName`	The key of the integer to retrieve.
`-> iDefaultValue`	The default integer to return if no integer can be retrieved for the specified key.
`<- oValue`	The buffer to receive the integer.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values and sets `oValue` to the value specified by `iDefaultValue`.

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
kUserMgrParamErr
```

This function can also return File Manager errors.

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetInteger`

New UmGetLibVersion

Purpose Retrieves the version of the User Manager API.

Prototype

```
OSStatus UmGetLibVersion (UInt16 *oVersionMajor,
UInt16 *oVersionMinor)
```

Parameters

`<- oVersionMajor`	The major version number.
`<- oVersionMinor`	The minor version number.

Result Returns `kUserMgrNoErr` if successful, otherwise returns one of the following error codes:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
```

Comments Call the `UmGetLibVersion` function to determine the version of the User Manager API that you are using before calling any of its functions. See "User Manager API Versions" on page 233.

NOTE: This reference documents version 1.0 of the User Manager API.

Compatibility User Manager version: 4.0 and later.

New

UmGetSlotCount

Purpose Retrieves the number of expansion slots on the handheld for the specified user.

Prototype

```
OSStatus UmSlotGetSlotCount (PalmUserID iUserID,
UInt16 *oSlotCount);
```

Parameters

`-> iUserID`	The ID of the user. You may use `kCurrentPalmUserID`.
`<- oSlotCount`	A pointer to a `UInt16` to hold the number of slots on the handheld. If the handheld has no expansion slots, the value it points to is 0.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
```

This function can also return File Manager errors.

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments HotSync Manager retrieves this information from the handheld at the beginning of each HotSync operation and saves it for the corresponding user in the user information store on the desktop. This function simply passes back the saved value. Therefore this value may not be accurate for the next HotSync operation, because the user may have changed or updated the handheld.

Example

```
UInt16 numSlots;
OSStatus retVal = UmGetSlotCount (userID, &numSlots);
```

Compatibility User Manager version: All.
Palm OS version: 4.0 or later.

New

UmGetSlotFolderSpec

Purpose Retrieves the `FSSpec` of the specified slot's folder inside the "Files to Install" folder.

Prototype

```
OSStatus UmGetSlotFolderSpec (PalmUserID iUserID,
UInt16 iSlot, FSSpec *oSlotFolderSpec);
```

Parameters

-> `iUserID`	The ID of the user. You may use `kCurrentPalmUserID`.
-> `iSlot`	Zero-based slot index.
<- `oSlotFolderSpec`	A pointer to an `FSSpec` to hold the `FSSpec` of the folder for the specified slot.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrInvalidSlotIndexErr
```

This function can also return File Manager errors.

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.
Palm OS version: 4.0 or later.

New UmGetString

Purpose Retrieves a string value from a key in the specified user's area of the users data store.

Prototype

```
OSStatus UmGetString (PalmUserID iUserID,
CFStringRef iSectionName, CFStringRef iKeyName,
CFStringRef iDefaultValue, CFStringRef *oValue);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iSectionName`	The section name in the specified user's area of the users data store.
`-> iKeyName`	The key of the string to retrieve.
`-> iDefaultValue`	The default string to return if no string can be retrieved for the specified key.
`<- oValue`	The buffer to receive the string.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error codes.

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
kUserMgrParamErr
```

This function can return File Manager and Core Foundation errors.

IMPORTANT: The caller is responsible for releasing the `oValue` result by calling `CFRelease`.

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments If the key specified by `iKeyName` doesn't exist, `UmGetString` returns `iDefaultValue`.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetString`

New UmGetUserByDirName

Purpose Returns the user ID for the user matching a given folder.

Prototype

```
OSStatus UmGetUserByDirName (CFStringRef iDirName,
PalmUserID *oUserID);
```

Parameters

`-> iDirName`	The name of the directory whose corresponding user ID is to be returned.
`<- oUserID`	A pointer to a `PalmUserID` to receive a user ID.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetIDFromDirectory`,

New UmGetUserCount

Purpose Returns the number of users in the users data store.

Prototype

```
OSStatus UmGetUserCount (UInt16 *oUserCount);
```

Parameters

`<- oUserCount`	A pointer to a `UInt16` buffer. On return, this buffer contains the number of users in the users data store.

Result If successful, stores a value >=0 in `oUserCount` that is the number of users in the users data store, and returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

New UmGetUserDataLastModDate

Purpose Returns the date and time of the last time the user data file was modified.

Prototype

```
CFAbsoluteTime UmGetUserDataLastModDate ();
```

Parameters If successful, returns the modification date and time of the `Palm Users` file.

If unsuccessful, returns the date 1-Jan-1904.

Comments User Manager version: All.

New UmGetUserID

Purpose Returns a user ID from the users data store by index.

Prototype

```
OSStatus UmGetUserID (UInt16 iIndex,
PalmUserID *oUserID);
```

Parameters

`-> iIndex`	A zero-based index that specifies a user in the users data store.
`<- oUserID`	A pointer to a `PalmUserID` to receive a user ID.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrInvalidUserIndexErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetIDFromDirectory`, `UmGetUserNameByID`

New UmGetUserName

Purpose Retrieves a user name in the users data store by index.

Prototype

```
OSStatus UmGetUserName (UInt16 iIndex,
CFStringRef *oUserName);
```

Parameters

`-> iIndex`	A zero-based index that specifies a user in the users data store.

`<- oUserName`	`CFString` reference to the name of the user.

Result If successful, returns `kUserMgrNoErr` and a `CFStringRef` to the user's name.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrInvalidUserIndexErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

IMPORTANT: In keeping with Apple's convention, the caller does **not** own the returned `CFStringRef` and should not call `CFRelease` on it.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetUserNameByID`

New UmGetUserNameByID

Purpose Retrieves a user name in the users data store by user ID.

Prototype

```
OSStatus UmGetUserNameByID (UInt16 iUserID,
CFStringRef *oUserName);
```

Parameters

`-> iUserID`	A user ID that specifies a user in the users data store. You may specify `kCurrentPalmUserID`.
`<- oUserName`	`CFString` reference to the name of the user.

Result If successful, returns `kUserMgrNoErr` and a `CFStringRef` to the user's name.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrInvalidUserIndexErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

IMPORTANT: In keeping with Apple's convention, the caller does **not** own the returned `CFStringRef` and should not call `CFRelease` on it.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetUserNameByID`

New UmGetUserPassword

Purpose Retrieves the encrypted user password for the specified user ID.

Prototype

```
OSStatus UmGetUserPassword (PalmUserID iUserID,
CFStringRef *oUserPassword);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`<- oUserPassword`	On return, contains the user's password.

Result If successful, `oUserPassword` is filled with a `CFStringRef` to the encrypted user password and `kUserMgrNoErr` is returned.

If unsuccessful, returns one of the following error code values.

```
kUserMgrCorruptUsersFileErr
kUserMgrInvalidUserIndexErr
kUserMgrNoUserPasswordErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmValidatePassword`

New UmGetUserPermSyncPreferences

Purpose Retrieves a conduit's permanent synchronization preferences for the specified user ID.

Prototype

```
OSStatus UmGetUserPermSyncPreferences
(PalmUserID iUserID,
ConduitCreator iConduitCreator, UmUserSyncAction
*oSyncPrefs);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iConduitCreator`	The creator ID of the conduit to access.
`<- oSyncPrefs`	A pointer to a `UmUserSyncAction` to receive the synchronization preferences defined by User Synchronization Action (UmUserSyncAction) Constants.

Result If successful, stores one of the User Synchronization Action (UmUserSyncAction) Constants into oSyncPrefs and returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrPrefNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetUserPermSyncPreferences`, `UmDeleteUserPermSyncPreferences`

New

UmGetUsersConduitsDirectory

Purpose Returns the `VRefNum` and `DirID` of the specified HotSync user's conduits directory.

Prototype

```
OSStatus UmGetUsersConduitsDirectory
(PalmUserID iUserID, SInt16 iDomain,
SInt16 *oConduitsVRefNum, SInt32 *oConduitsDirID);
```

Parameters

`-> iUserID`	User ID of the user whose directory information is to be returned.
`-> iDomain`	The Folder Domain to look in. See Chapter 9, "Installing Conduits," on page 111 in the *C++ Sync Suite Companion for Macintosh* for details.
`<- oConduitsVRefNum`	A buffer to receive the volume reference number.
`<- oConduitsDirID`	A buffer to receive the directory ID number.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrParamErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

TIP: This function takes the place of hunting down the active Serial Monitor in the desktop database and finding its directory.

Currently there is no conduits folder associated with a given user, so this function is equivalent to `UmGetGlobalConduitsDirectory`; however, in a future release, there is likely to be a conduits folder on a per HotSync user basis.

Compatibility User Manager version: All.

See Also `UmGetUserID`

New UmGetUserTempSyncPreferences

Purpose Retrieves a conduit's temporary synchronization preferences for the specified user ID.

Prototype

```
OSStatus UmGetUserTempSyncPreferences
(PalmUserID iUserID,
ConduitCreator iConduitCreator, UmUserSyncAction
*oSyncPrefs);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iConduitCreator`	The creator ID of the conduit to access.
`<- oSyncPrefs`	A pointer to a `UmUserSyncAction` to receive the synchronization preferences defined by User Synchronization Action (UmUserSyncAction) Constants.

Result If successful, stores in `oSyncPrefs` one of the User Synchronization Action (UmUserSyncAction) Constants and returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrPrefNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetUserTempSyncPreferences`, `UmRemoveUserTempSyncPreferences`, `UmDeleteUserTempSyncPreferences`

New

UmIsUserInstalled

Purpose Determines whether the specified user is "installed."

Prototype

```
OSStatus UmIsUserInstalled (PalmUserID iUserID,
Boolean *oIsInstalledUser);
```

Parameters

`-> iUserID` The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.

`<- oIsInstalledUser`
`true` if the specified user is installed; `false` if not installed.

Result If the user is an installed user, places `true` in `oIsInstalledUser`, and returns `kUserMgrNoErr`.

If the user is not an installed user, places `false` in `oIsInstalledUser` and returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments "Installed" users are those who have completed at least one HotSync operation so that their user ID is on both the desktop computer and a handheld. Users created on the desktop, but who have never completed a HotSync operation, are not "installed" users. They become "installed" users if they synchronize a handheld with the same user ID as is on the desktop.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmSetUserInstall`

New

UmIsUserProfile

Purpose Determines, by the user ID, whether a user is a profile user.

Prototype

```
OSStatus UmIsUserProfile (PalmUserID iUserID,
Boolean *oIsProfileUser);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`<- oIsProfileUser`	`true` if the specified user is a profile user; `false` if not a profile user.

Result If the user is a profile user, places `true` in `oIsProfileUser` and returns `kUserMgrNoErr`.

If the user is not a profile user, places `false` in `oIsProfileUser` and returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments A *user profile* is a special account set up on the desktop computer that installs applications and data on a handheld that does not have a user name and user ID — that is, a handheld that has never completed a HotSync operation or has just been hard reset. This feature allows many handhelds to be preloaded in the same way before they are assigned a user name and user ID on the first synchronization.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmIsUserProfile`

New

UmReloadUserData

Purpose Reloads the users data to ensure that it's up to date in case it has changed since the last time it was loaded.

Prototype `OSStatus UmReloadUserData ();`

Parameters If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns `kUserMgrParamErr`

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments User Manager version: All.

New UmRemoveUserTempSyncPreferences

Purpose Removes the specified conduit's temporary synchronization preferences for the specified user ID.

NOTE: This function deletes only one conduit's temporary synchronization preferences. Contrast it with `UmDeleteUserTempSyncPreferences`, which deletes the temporary preferences for *all* the user's conduits.

Prototype

```
OSStatus UmRemoveUserTempSyncPreferences
(PalmUserID iUserID, ConduitCreator
iConduitCreator);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iConduitCreator`	The creator ID of the conduit to access.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFile
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments The `UmRemoveUserTempSyncPreferences` function unsets the temporary synchronization preference. The result is the same as if the user has never clicked HotSync Manager's **Custom** > **Change** option and altered a synchronization preference.

Compatibility User Manager version: All.

See Also UmGetUserID, UmSetUserTempSyncPreferences, UmGetUserTempSyncPreferences, UmDeleteUserTempSyncPreferences

New

UmSetInteger

Purpose Sets an integer value to a key in the specified user's area of the users data store.

Prototype

```
OSStatus UmSetInteger (PalmUserID iUserID,
CFStringRef iSectionName, CFStringRef iKeyName,
SInt32 iValue);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iSectionName`	The section name in the specified user's area of the users data store.
`-> iKeyName`	The key of the integer to set.
`-> iValue`	The integer to write to the key in the specified user's area of the users data store.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
kUserMgrParamErr
```

This function can return File Manager errors.

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also UmGetUserID, UmGetInteger

New UmSetString

Purpose Sets a string value to a key in the specified user's area of the users data store.

Prototype

```
OSStatus UmSetString (PalmUserID iUserID,
CFStringRef iSectionName, CFStringRef iKeyName,
CFStringRef iValue);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iSectionName`	The section name in the specified user's area of the users data store.
`-> iKeyName`	The key of the string to set.
`-> iValue`	The string to write to the key in the specified user's area of the users data store.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
kUserMgrParamErr
```

This function can return File Manager and Core Foundation errors.

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmGetString`

New UmSetUserDirectory

Purpose Sets the directory name of the specified user ID.

Prototype

```
OSStatus UmSetUserDirectory (PalmUserID iUserID,
CFStringRef iUserDirectoryName);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iUserDirectoryName`	A `CFStringRef` that contains the user directory name to set.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrSaveErr
```

For more information about the error codes, see “User Manager Error Codes” on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmCopyUserDirectory`

New UmSetUserInstall

Purpose Sets or clears the "installed" flag of the specified user ID.

Prototype

```
OSStatus UmSetUserInstall (PalmUserID iUserID,
Boolean iIsInstalledUser);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iIsInstalledUser`	A Boolean value. If this is `true`, the user is set as an "installed" user; if this is `false`, the user is not set as an "installed" user.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrSaveErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Comments "Installed" users are those who have completed at least one HotSync operation so that their user ID is on both the desktop computer and a handheld. Users created on the desktop, but who have never completed a HotSync operation, are not "installed" users. They become "installed" users if they synchronize a handheld with the same user ID as is on the desktop.

Compatibility User Manager version: All.

See Also `UmGetUserID`, UmIsUserInstalled

New

UmSetUserNameByID

Purpose Sets the user name of the specified user ID.

Prototype

```
OSStatus UmSetUserNameByID (PalmUserID iUserID,
CFStringRef iUserName);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iUserName`	A `CFStringRef` that indicates the user name to set.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFile
kUserMgrParamErr
kUserMgrUserNotFoundErr
kUserMgrInvalidUserNameErr
kUserMgrSaveErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmGetUserName`

New UmSetUserPermSyncPreferences

Purpose Sets a conduit's permanent synchronization preferences for the specified user ID.

Prototype

```
OSStatus UmSetUserPermSyncPreferences
(PalmUserID iUserID,
ConduitCreator iConduitCreator,
UmUserSyncAction iSyncPrefs);
```

Parameters

-> `iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
-> `iConduitCreator`	The creator ID of the conduit to access.
-> `iSyncPrefs`	The synchronization preference for this conduit and user. Use the flags described in "User Synchronization Action (UmUserSyncAction) Constants" on page 234.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmGetUserPermSyncPreferences`, `UmDeleteUserPermSyncPreferences`

New UmSetUserTempSyncPreferences

Purpose Sets a conduit's temporary synchronization preferences for the specified user ID.

Prototype

```
OSStatus UmSetUserTemoSyncPreferences
(PalmUserID iUserID,
ConduitCreator iConduitCreator, UmUserSyncAction
iSyncPrefs);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`-> iConduitCreator`	The creator ID of the conduit to access.
`-> iSyncPrefs`	The synchronization preferences to use. Use the flags described in "User Synchronization Action (UmUserSyncAction) Constants" on page 234.

Result If successful, returns `kUserMgrNoErr`.

If unsuccessful, returns one of the following error code values:

```
kUserMgrCorruptUsersFileErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmGetUserTempSyncPreferences`, `UmRemoveUserTempSyncPreferences`, `UmDeleteUserTempSyncPreferences`

New UmValidatePassword

Purpose Compares the plaintext password specified to the specified HotSync user's encrypted password.

Prototype

```
OSStatus UmValidatePassword (PalmUserID iUserID,
CFStringRef iPassword);
```

Parameters

`-> iUserID`	The user ID, which specifies the user to reference in the users data store. You may specify `kCurrentPalmUserID`.
`<- iPassword`	The plaintext password to compare against the user's encrypted password.

Result Returns `kUserMgrNoErr` if the password is a match. Otherwise returns one of the following error code values.

```
kUserMgrNoUserPasswordErr
kUserMgrInvalidUserPasswordErr
kUserMgrUserNotFoundErr
```

For more information about the error codes, see "User Manager Error Codes" on page 273.

Compatibility User Manager version: All.

See Also `UmGetUserID`, `UmGetUserPassword`

User Manager Error Codes

Table 6.1 describes the negative error codes that the User Manager functions can return. These error codes are declared in the `UserMgr.h` header file.

Table 6.1 User Manager Error Codes

Error Code	Description
`kUserMgrParamErr`	An invalid parameter value was specified.
`kUserMgrInvalidUserNameErr`	An invalid user name was specified (such as one that's longer than 20 characters).
`kUserMgrUserAlreadyExistsErr`	The specified user already exists.
`kUserMgrSaveErr`	Error saving changes to users data file.
`kUserMgrCantCreateNewFileErr`	An error occurred attempting to create a new users data file.
`kUserMgrNoUsersErr`	No users exist.
`kUserMgrInvalidUserErr`	The specified user is invalid.
`kUserMgrInvalidUserIndexErr`	The specified user index isn't valid.
`kUserMgrHSNotInstalledErr`	HotSync Manager isn't installed.
`kUserMgrNoUsersDataFileErr`	The users data file can't be found.
`kUserMgrNoDirectoryErr`	The user's directory couldn't be found.
`kUserMgrUserNotFoundErr`	The specified user couldn't be found.
`kUserMgrDirectoryInUseErr`	The specified directory is already in use.
`kUserMgrInvalidUserDirErr`	The user directory is not valid.

Table 6.1 User Manager Error Codes *(continued)*

Error Code	Description
`kUserMgrCorruptUsersFileErr`	The users data file is corrupt.
`kUserMgrPrefNotFoundErr`	The preference for a conduit wasn't found.
`kUserMgrNoUserPasswordErr`	The user doesn't have a password.
`kUserMgrInvalidUserPasswordErr`	The user's password isn't valid.
`kUserMgrSyncIniFileFormatErr`	The `sync.ini` file is not properly formatted.
`kUserMgrSyncIniNoSuchSectionErr`	The `sync.ini` file does not contain the specified section.

A

Private Functions

This appendix lists the private functions embedded in the HotSync® Manager library for internal use only. These are private, system functions that you must not use in your conduits. This list is documented here only because you may see these function names in the source code and may not recognize the implications of using them.

WARNING! Do not use these functions in your code! Doing so can interfere with the HotSync Manager application and corrupt all data on the handheld.

- `SyncDeInit`
- `SyncEndOfSync`
- `SyncInit`
- `SyncLoopBackTest`
- `SyncPreSendCmd`
- `SyncReadProdCompInfo`
- `SyncReadNetSyncInfo`
- `SyncWriteNetSyncInfo`
- `SyncWriteUserID`

Obsolete Functions

This section lists the functions that are no longer available in the Conduit Development Kit.

Function Name	Notes
`SyncCallApplication`	Available in Palm OS® versions earlier than version 2.0. If you call this function on a handheld running a version 2.0 or later of the Palm OS, your application receives an "illegal request" error.

Index